Solving Hierarchical Soft Constraints with an SMT Solver

Hiroshi Hosobe
Faculty of Computer and Information Sciences, Hosei University
3-7-2 Kajino-cho, Koganei-shi, Tokyo 184-8584, Japan
hosobe@acm.org

ABSTRACT
Constraints allow the declarative specification of various problems in many fields. In particular, constraint hierarchies that enable soft constraints with hierarchical preferences are useful for programming interactive graphical applications. However, it is still difficult to handle constraint hierarchies with nonlinear constraints. This paper proposes an algorithm for solving constraint hierarchies possibly with nonlinear constraints. Instead of directly solving a constraint hierarchy, it successively generates and solves ordinary constraint problems by using an external SMT solver. The results of our experiments show that the algorithm is able to find accurate constraint hierarchy solutions.

CCS Concepts
• Theory of computation→Constraint and logic programming; • Mathematics of computing →Solvers

Keywords
Constraint solving; Soft constraint; Constraint hierarchy; SMT

1. INTRODUCTION
Constraints allow the declarative specification of various problems in many fields such as artificial intelligence, software, and computer-aided design. In particular, since constraints can naturally express positional relationships among visual objects in graphical applications, constraints have been studied in the fields of user interfaces and computer graphics since its infancy [1] until recently [2][3].

Soft constraints are important for constraint-based interactive graphical applications. It is necessary to provide each visual object with various constraints that represent, for example, its possible positional range on a screen, its relative positional relationship with other objects, and the dragging of it with a mouse. Therefore, it is difficult for programmers to consistently specify all constraints without using soft constraints.

Constraint hierarchies [4] are widely used as a theoretical framework for soft constraints. In a constraint hierarchy, each constraint is associated with a hierarchical preference called a *strength*, and solutions are determined to satisfy as many strong constraints as possible. There has been much research on algorithms for solving constraint hierarchies for interactive graphical applications. In particular, there are efficient algorithms for solving constraint hierarchies consisting of linear constraints. However, it is still difficult to handle constraint hierarchies with nonlinear constraints.

In this paper, we propose an algorithm for solving constraint hierarchies possibly with nonlinear constraints. Instead of directly solving a constraint hierarchy, our algorithm successively generates and solves ordinary constraint problems by using an external SMT solver. The results of our experiments show that the algorithm is able to find accurate constraint hierarchy solutions based on the criterion called least-squares-better (LSB).

The rest of this paper is organized as follows. Section 2 describes previous work related to the proposed algorithm. Next, Section 3 provides a brief introduction to constraint hierarchies and their notations. Then Section 4 proposes our algorithm, and Section 5 gives its implementation. Section 6 presents the results of our experiments, and Section 7 discusses the algorithm. Finally, Section 8 provides conclusions and future work.

2. RELATED WORK
Most of early solvers for constraint hierarchies [5][6][7][8] treated dataflow constraints by using a graph-theoretic approach. Algorithms of this kind were limited in application domains because of their insufficient ability to process simultaneous constraints and inequality constraints.

Linear constraint solvers [9][10][11][12][13] appropriately process simultaneous constraints and inequality constraints in constraint hierarchies. In particular, Cassowary [9] is widely used as an internal solver of Apple Auto Layout [14]. However, these solvers are unable to treat nonlinear constraints including geometric constraints.

To solve constraint hierarchies including nonlinear constraints, approximate algorithms have been proposed [15][16][17]. For example, Chorus [15] approximately finds layouts of geometric objects that slightly differ from correct ones, usually by several pixels on computer screens. In fact, solving the constraint hierarchy consisting of strong $x = 0$ and medium $x = 100$, it obtains $x = 3.0303\cdots$ in a typical setting.

There are more accurate algorithms for solving constraint hierarchies with nonlinear constraints, for example, based on hierarchical least squares [18] and hierarchical Lagrange multipliers [19]. However, to the author's knowledge, there are still no solvers for constraint hierarchies with nonlinear constraints that are used in practical applications.

Satisfiability modulo theories (SMT) [20] is gaining attention as a general method for solving various kinds of constraints. Z3 [21] is an example of an SMT solver. In this paper, we attempt to utilize the general power of an SMT solver to solve constraint hierarchies.

ICCAE 2020, February 14–16, 2020, Sydney, NSW, Australia

ACM ISBN 978-1-4503-7678-5/20/02…$15.00
https://doi.org/10.1145/3384613.3384654

3. CONSTRAINT HIERARCHIES

We treat the problem of finding a single solution to a constraint hierarchy including nonlinear constraints over real-number domains, based on the solution criterion called *least-squares-better* (LSB) [4]. In a constraint hierarchy, each constraint is associated with a hierarchical preference called a strength. A constraint hierarchy consists of a finite number of levels, where the top level contains *required* (or hard) constraints that must always be satisfied, and lower levels contain *preferential* (or soft) constraints that can be relaxed. Strengths are often symbolically represented as required, strong, medium, and weak. The LSB solution set consists of the solutions obtained by successively computing the least-squares sums of the errors of equal-strength constraints from the strongest to the weakest.

We use the following notations. A strength is represented as an integer between 0 and some positive integer l. Strength 0 indicates that the associated constraints are required, and non-zero strengths indicate that the associated constraints are preferential; intuitively, a strength represented as a larger integer k means a weaker preference. A constraint hierarchy is represented as $H = (H_0, H_1, H_2, \ldots, H_l)$, where each H_k indicates the multiset of the constraints with strength k. (Since a constraint hierarchy may include multiple occurrences of the same constraints, a multiset is used instead of an ordinary set.) The j-th constraint at level k of H is represented as c_j^k. The variables that appear in H are represented as a vector $\boldsymbol{x}$, each component x_i of which indicates a variable.

4. PROPOSED ALGORITHM

We propose an algorithm for finding LSB solutions to constraint hierarchies by using an SMT solver. Instead of directly solving a constraint hierarchy, our algorithm successively generates and solves constraint problems. To solve such constraint problems, we use an SMT solver. Note that the SMT solver needs to handle neither soft constraints nor objective functions.

Our algorithm optimizes each level of a constraint hierarchy from the strongest to the weakest. Consider, for instance, a constraint hierarchy with three levels, required, strong, and weak. Intuitively, the algorithm works as follows.

1. First, it solves the problem consisting of all the required constraints. Since this is a simple constraint problem without soft constraints and objective functions, it can be handled with an SMT solver. If there is no solution, the algorithm reports it (which is correct from the viewpoint of the constraint hierarchy theory).
2. Next, it minimizes the objective function corresponding to strong constraints while satisfying the required constraints. For this purpose, it constructs the objective function by summing the squares of the differences of the left- and right-hand sides of the strong constraints. Also, it minimizes the objective function by performing binary search for its minimum value.
3. Finally, it minimizes the objective function corresponding to weak constraints while satisfying the required constraints and preserving the minimality of the objective function corresponding to the strong constraints, in the same way as the minimization step for the strong constraints.

Figure 1 presents the general description of the proposed algorithm. It takes as input a constraint hierarchy H with variables $\boldsymbol{x}$. If it finds an LSB solution to H, it returns a variable assignment for $\boldsymbol{x}$ that represents the solution; otherwise, it returns an empty variable assignment. A variable assignment θ for $\boldsymbol{x}$ is a function from $\boldsymbol{x}$ to real numbers; given a variable x_i in $\boldsymbol{x}$, $\theta(x_i)$ represents the value of x_i. Internally, the algorithm uses a function "solve" that solves a constraint problem C for $\boldsymbol{x}$, denoted as $\mathrm{solve}(C, \boldsymbol{x})$, for which it calls an external SMT solver. It also uses a function "error" that represents the error of a constraint in two ways: when denoted as $\mathrm{error}(c_j^k, \boldsymbol{x})$, the error of constraint c_j^k is presented as a function of $\boldsymbol{x}$; when denoted as $\mathrm{error}\left(c_j^k, \theta(\boldsymbol{x})\right)$, the error of c_j^k for variables $\boldsymbol{x}$ whose values are assigned by θ is computed as a real number. Typically, the error of a constraint is computed as the absolute value of the difference between its left- and right-hand sides (which is also called the violation or residual of the constraint). The algorithm internally uses a constant, very small positive real number ε to determine whether the computed upper and lower bounds are close enough during the binary search.

```
 1: θ ← solve(H_0, x)
 2: if no such θ was found then
 3:     return empty assignment
 4: end if
 5: C ← H_0
 6: for k = 1 to l do
 7:     b_low ← 0
 8:     b_up ← Σ_{c_j^k ∈ H_k} error(c_j^k, θ(x))^2
 9:     while b_up − b_low > ε do
10:         c ← (Σ_{c_j^k ∈ H_k} error(c_j^k, x)^2 ≤ (b_low + b_up)/2)
11:         θ′ ← solve(C ∪ {c}, x)
12:         if such θ′ was found then
13:             θ ← θ′
14:             b_up ← Σ_{c_j^k ∈ H_k} error(c_j^k, θ(x))^2
15:         else
16:             b_low ← (b_low + b_up)/2
17:         end if
18:     end while
19:     c ← (Σ_{c_j^k ∈ H_k} error(c_j^k, x)^2 ≤ b_up)
20:     C ← C ∪ {c}
21: end for
22: return θ
```

Figure 1: Algorithm for solving a constraint hierarchy H with variables $\boldsymbol{x}$.

The algorithm works as follows. First, it processes the required constraints at lines 1 to 4. Next, it processes the preferential constraints from the strongest to the weakest at lines 5 to 21. Inside the "for" loop from line 6, it processes the constraints with particular strength k. At lines 7 to 18, it computes the upper bound b_{up} of the least-squares sum of the errors of the constraints with strength k by binary search. After this, it generates a new constraint at line 19 that limits the least-squares sum by the computed upper bound, and preserves it at line 20 for later computation. Finally, it returns a variable assignment as a solution at line 22.

5. IMPLEMENTATION

We implemented the proposed algorithm in Python 3.7.3 by using version 4.8.7 of the Z3 SMT solver [21]. Its application programming interface is similar to that of Z3. It provides a class named `ch.Solver` that implements the proposed algorithm. As shown in an example program in Figure 2, required and

preferential constraints can be added to an instance of **`ch.Solver`**, and a solution is computed by calling the **`solve`** method of **`ch.Solver`**. The current implementation internally uses $\varepsilon = 10^{-6}$ for the binary search in the algorithm.

```
import z3
import ch
solver = ch.Solver()
x = z3.Real('x')
y = z3.Real('y')
solver.add(y >= x * x)
solver.add(x >= -1)
solver.add_strong(x == -2)
solver.add_weak(y == -1)
solution = solver.solve()
```

Figure 2: Example program for solving a constraint hierarchy with our implementation of the proposed algorithm.

6. EXPERIMENTS

In this section, we show the results of the three experiments that we conducted to evaluate the proposed algorithm.

6.1 Linear Constraints

For the first experiment, we use the following constraint hierarchy that consists of five linear constraints:

required $y \geq -x$
required $y \geq x$
required $x \geq -1$
strong $x = -2$
weak $y = -1$.

Among the constraint hierarchies that we present in this paper, this is the simplest in the sense of kinds of constraints. It simplifies the example given in [19] by using two linear inequality constraints instead of a nonlinear inequality constraint.

Theoretically, there is only one LSB solution $(x, y) = (-1, 1)$ to this constraint hierarchy, which is illustrated in Figure 3. Any solution must be inside or on the border of the yellow region formed by the three required constraints. To maximally satisfy strong $x = -2$, which is the next strongest, it must satisfy $x = -1$. To maximally satisfy weak $y = -1$, the solution $(x, y) = (-1, 1)$ is uniquely determined.

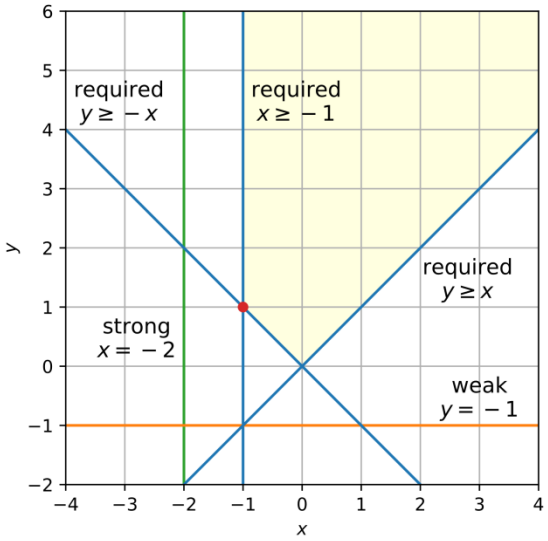


Figure 3: Constraint hierarchy consisting of linear constraints.

The program that solves this constraint hierarchy with our implementation of the proposed algorithm is given in Figure 4(a). Its execution obtained $(x, y) = \left(-\frac{16777215}{16777216}, 1\right)$, which is very close to the theoretical LSB solution $(-1, 1)$. The average time of ten executions of the solution computation was 0.326 seconds on a 1.6 GHz Core i5-8250U processor running Windows 10.

```
import z3
import ch
solver = ch.Solver()
x = z3.Real('x')
y = z3.Real('y')
solver.add(y >= -x)
solver.add(y >= x)
solver.add(x >= -1)
solver.add_strong(x == -2)
solver.add_weak(y == -1)
solution = solver.solve()
```

(a)

```
import z3
solver = z3.Optimize()
x = z3.Real('x')
y = z3.Real('y')
solver.add(y >= -x)
solver.add(y >= x)
solver.add(x >= -1)
solver.add_soft(x == -2, 10)
solver.add_soft(y == -1, 1)
solver.check()
solution = solver.model()
```

(b)

```
import z3
solver = z3.Optimize()
x = z3.Real('x')
y = z3.Real('y')
solver.add(y >= -x)
solver.add(y >= x)
solver.add(x >= -1)
e1 = z3.Real('e1')
solver.add(x == -2 + e1)
e2 = z3.Real('e2')
solver.add(y == -1 + e2)
solver.minimize(10 * e1 * e1 + e2 * e2)
solver.check()
solution = solver.model()
```

(c)

Figure 4: Programs for solving a constraint hierarchy with (a) the proposed algorithm, (b) a simple use of Z3, and (c) another simple use of Z3.

For comparison, we show the results of executing two other programs based on simple uses of the Z3 SMT solver. One program adopts Z3's soft constraint facility by using the **`z3.Optimize`** class and its **`add_soft`** method, as shown in Figure 4(b). The other adopts Z3's optimization facility by using **`z3.Optimize`** and its **`minimize`** method, as shown in Figure 4(c). The executions of both programs obtained $(x, y) = (0, 0)$, which is far from the theoretical LSB solution $(-1, 1)$ to the given constraint hierarchy. These results suggest that such simple uses of Z3 are not usable to constraint hierarchies.

6.2 Nonlinear and Linear Constraints

For the second experiment, we use the following constraint hierarchy that consists of one nonlinear and three linear constraints:

required	$y \geq x^2$
required	$x \geq -1$
strong	$x = -2$
weak	$y = -1$.

This was taken from [19] (where the two inequality constraints were represented as equations by introducing slack variables). It is a more complex version of the constraint hierarchy shown in Subsection 6.1; it uses a nonlinear constraint $y \geq x^2$ instead of two linear constraints $y \geq -x$ and $y \geq x$.

It has only one theoretical LSB solution $(x, y) = (-1, 1)$, which is illustrated in Figure 5. How the solution is determined can be understood in the same way as that of the constraint hierarchy in Subsection 6.1.

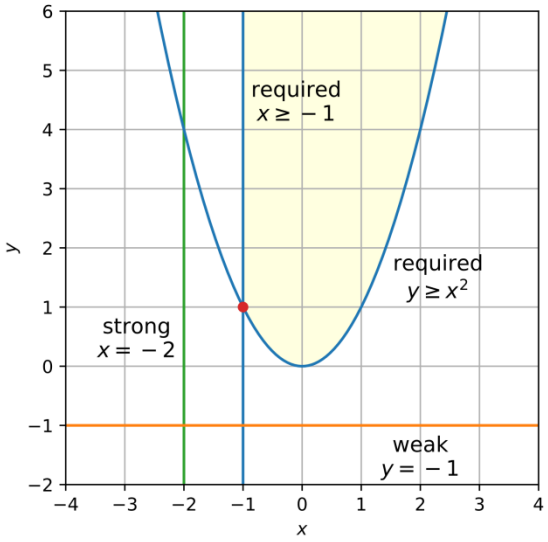


Figure 5: Constraint hierarchy consisting of a nonlinear constraint and linear constraints.

The program that solves this constraint hierarchy with the proposed algorithm is given in Figure 2. The execution of the program obtained $(x, y) = (-1, 1)$, which is exactly the same as the theoretical LSB solution. The average time of ten executions of the solution computation was 0.259 seconds.

6.3 Graphical Constraints

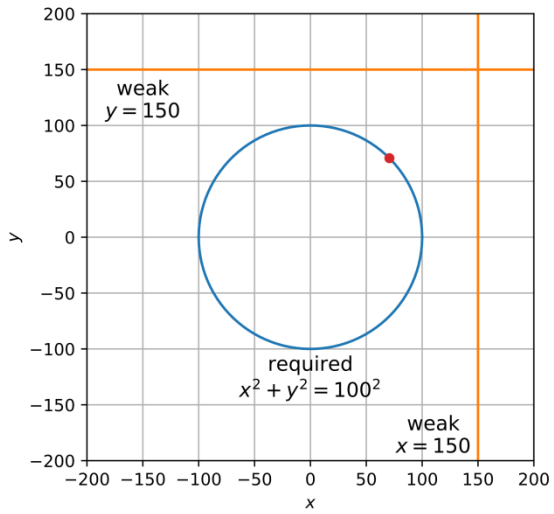
For the third experiment, we use a problem of graphical layout that consists of two constraint hierarchies. This problem was taken from [18]. It first treats the following constraint hierarchy:

required	$x^2 + y^2 = 100^2$
weak	$x = 150$
weak	$y = 150$.

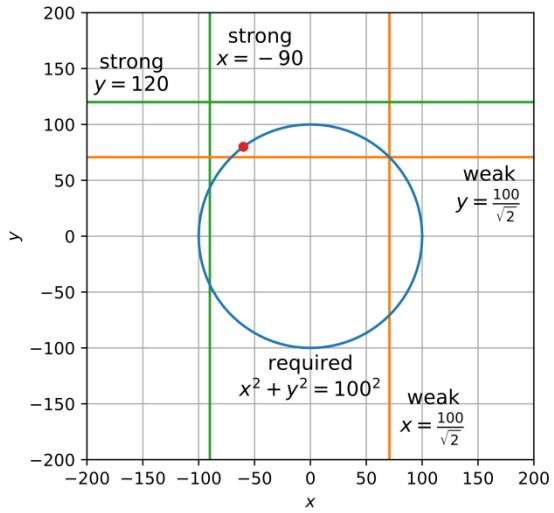
Next, it treats the following constraint hierarchy:

required	$x^2 + y^2 = 100^2$
strong	$x = -90$
strong	$y = 120$
weak	$x = 100/\sqrt{2}$
weak	$y = 100/\sqrt{2}$.

This problem indicates a typical use of strong and weak constraints in interactive graphical applications. It assumes a situation where a point (x, y) that is constrained to be on a circle is moved by a user. As shown in Figure 6(a), the first constraint hierarchy obtains the initial position of the point in such a way that it will be as close to $(150, 150)$ (which can be regarded as the default position) as possible. The theoretical LSB solution to this constraint hierarchy is $(x, y) = \left(100/\sqrt{2}, 100/\sqrt{2}\right) = (70.7\cdots, 70.7\cdots)$. After this, the second constraint hierarchy modifies the position of the point, as shown in Figure 6(b), in such a way that it will be as close to $(-90, 120)$ (which indicates the position to which the user tries to move the point) as possible while still trying to satisfy the weak preference for its closeness to the previous position $\left(100/\sqrt{2}, 100/\sqrt{2}\right)$ (which can be regarded as the revised default position). The theoretical LSB solution to this constraint hierarchy is $(x, y) = (-60, 80)$.



(a)



(b)

Figure 6: Constraint hierarchies consisting of graphical constraints for (a) an initial layout and (b) a modified layout.

The execution of the program for solving the first constraint hierarchy obtained $(x, y) = \left(\frac{42426231301022359}{600000000000000}, 70.7109707343\right)$, which is very close to the theoretical LSB solution $(100/$

$\sqrt{2}, 100/\sqrt{2})$. The average time of ten executions of the solution computation was 0.304 seconds. The execution of the program for solving the second constraint hierarchy obtained $(x, y) = (-60, 80)$, which is exactly the same as the theoretical LSB solution. The average time of ten executions of the solution computation was 0.835 seconds.

7. DISCUSSION

The results of the experiments presented in this paper showed that the proposed algorithm was accurate in computing solutions to constraint hierarchies. As described in Section 2, it is difficult for approximate methods to obtain such accurate solutions. The characteristic of the proposed algorithm when compared with other accurate methods is its simplicity. Although hierarchical least squares [18] is accurate, its algorithm is complex and is difficult to implement. Also, although the method of hierarchical Lagrange multipliers [19] is simpler than hierarchical least squares, its algorithm is still more complex than the proposed algorithm. We achieved the simplicity of the proposed algorithm by utilizing the general power of an SMT solver.

A primary problem with the proposed algorithm is its inefficiency. The experiment presented in Subsection 6.3 treated a typical use of preferential constraints in interactive graphical applications. In this experiment, the implementation of the proposed algorithm solved the first constraint hierarchy in 0.304 seconds and the second one in 0.835 seconds. Although solving the first hierarchy in such a length of time usually is acceptable, solving the second one in nearly one second can be a problem. This is because the second one represents a constraint hierarchy that repeatedly appears while a user is moving an object in an interactive graphical application. For such an application to be practical, the used constraint solver should solve such a hierarchy at most in 0.1 seconds. Therefore, the current implementation of the proposed algorithm has not yet achieved sufficient efficiency.

We treated LSB as the solution criterion for constraint hierarchies. Although LSB was used by previous arithmetic constraint hierarchy solvers [11][13][15][16][18][19], it is not an only solution criterion. In particular, weighted-sum-better (WSB) and locally-error-better (LEB) (which is less restrictive than WSB) also were used by previous solvers that handled linear and dataflow constraints including inequalities [7][9][10][12]. However, it is unclear whether these criteria are useful for nonlinear arithmetic constraints.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an algorithm for solving constraint hierarchies possibly with nonlinear constraints by using an SMT solver. The results of the experiments showed that the algorithm was able to find accurate constraint hierarchy solutions.

Our future work includes the development of interactive applications using the proposed algorithm and the examination of its practical utility. Also, it is necessary to improve the efficiency of the proposed algorithm.

9. ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number JP15KK0016.

10. REFERENCES

[1] I.E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proc. AFIPS Spring Joint Conf.*, pages 329–346, 1963.

[2] C. Zeidler, C. Lutteroth, W. Stuerzlinger, and G. Weber. The Auckland layout editor: An improved GUI layout specification process. In *Proc. ACM UIST*, pages 343–352, 2013.

[3] Y. Jiang, R. Du, C. Lutteroth, and W. Stuerzlinger. ORC layout: Adaptive GUI layout with OR-constraints. In *Proc. ACM CHI*, number 413, pages 1–12, 2019.

[4] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp Symbolic Comput.*, 5(3):223–270, 1992.

[5] B.N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Comm. ACM*, 33(1):54–63, 1990.

[6] H. Hosobe, K. Miyashita, S. Takahashi, S. Matsuoka, and A. Yonezawa. Locally simultaneous constraint satisfaction. In *Proc. PPCP Workshop*, volume 874 of *LNCS*, pages 51–62, 1994.

[7] A. Borning, R. Anderson, and B. Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. In *Proc. ACM UIST*, pages 129–136, 1996.

[8] B. Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way dataflow constraints. *ACM Trans. Prog. Lang. Syst.*, 18(1):30–72, 1996.

[9] H. Hosobe. A scalable linear constraint solver for user interface construction. In *Proc. CP*, volume 1894 of LNCS, pages 218–232, 2000.

[10] G.J. Badros, A. Borning, and P.J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Human Interact.*, 8(4):267–306, 2001.

[11] K. Marriott and S.S. Chok. QOCA: A constraint solving toolkit for interactive graphical applications. *Constraints*, 7(3–4):229–254, 2002.

[12] H. Hosobe. A simplex-based scalable linear constraint solver for user interface applications. In *Proc. IEEE ICTAI*, pages 793–798, 2011.

[13] N. Jamil, D. Needell, J. Müller, C. Lutteroth, and G. Weber. Kaczmarz algorithm with soft constraints for user interface layout. In *Proc. IEEE ICTAI*, pages 818–824, 2013.

[14] Apple Inc. *Auto Layout Guide*. 2011.

[15] H. Hosobe. A modular geometric constraint solver for user interface applications. In *Proc. ACM UIST*, pages 91–100, 2001.

[16] H. Hosobe. A geometric constraint library for 3D graphical applications. In *Proc. Smart Graphics*, pages 94–101. ACM, 2002.

[17] N. Hurst, K. Marriott, and P. Moulder. Dynamic approximation of complex graphical constraints by linear constraints. In *Proc. ACM UIST*, pages 191–200, 2002.

[18] H. Hosobe. Hierarchical nonlinear constraint satisfaction. In *Proc. ACM SAC*, pages 16–20, 2004.

[19] H. Hosobe. A hierarchical method for solving soft nonlinear constraints. In *Proc. SCSE*, volume 62 of *Procedia CS*, pages 378–384, 2015.

[20] D. Monniaux. A survey of satisfiability modulo theory. In *Proc. CASC*, volume 9890 of LNCS, pages 401–425, 2016.

[21] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.