Workshop on Reactive and Event-based Languages and Systems (REBLS) 2014
Co-located with SPLASH, October 21, 2014, Portland, Oregon, USA

Solving Interactive Logic Puzzles With Object-Constraints

An Experience Report Using Babelsberg/S for Squeak/Smalltalk

Maria Graber[1] Tim Felgentreff[2] Robert Hirschfeld[2]

Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
[1] maria.graber@student.hpi.uni-potsdam.de
[2] firstname.lastname@hpi.uni-potsdam.de

Alan Borning
University of Washington
Seattle, WA, USA
borning@cs.washington.edu

ABSTRACT

Logic puzzles such as Sudoku are described by a set of properties that a valid solution must have. Constraints are a useful technique to describe and solve for such properties. However, constraints are less suited to express imperative interactions in a user interface for logic puzzles, a domain that is more readily expressed in the object-oriented paradigm.

Object constraint programming provides a design to integrate constraints with dynamic, object-oriented programming languages. It allows developers to encode multi-way constraints over objects and object collections using existing, object-oriented abstractions. These constraints are automatically maintained at run-time.

In this paper we present an application of this design to logic puzzles in the Squeak/Smalltalk programming environment. We argue that our implementation facilitates event-driven applications with constraints on different parts of the system, by moving the burden to maintain the constraints from the developer to the runtime environment.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints*

General Terms

Languages

Keywords

Constraints, Object Constraint Programming, Constraint Imperative Programming, Babelsberg

1. INTRODUCTION

Logic puzzles are declarative. These puzzles can be solved without any world knowledge other than the rules of the puzzle and logical deduction techniques. A famous example is Sudoku. The rules of a logic puzzle describe properties that should be maintained while solving the puzzle. For example, in Sudoku, the properties are that each row, column, and box contain the numbers from 1 to 9 exactly once. The properties of a logic puzzle can be formulated as formal constraints, which a constraint solver can use to find one or more solutions or to check if a solution input by the user is valid [10].

User interface frameworks such as *Morphic* [13] are inherently imperative – the user interface consists of compositions of *Morphs* that have state and react to user input events. Morphic was first implemented in Self, with later implementations in Squeak [14] and JavaScript [17].

Babelsberg [6] is a design to integrate constraints into object-oriented languages in a way that allows programmers to dynamically create and satisfy constraints on objects. The design is a strict extension of the object-oriented semantics of the underlying host language. Babelsberg uses object-oriented method definitions to define constraints rather than a constraint domain-specific language (DSL) [15, 5, 16]. As a consequence, Babelsberg respects encapsulation and object-oriented abstractions. The design also supports solver features such as constraint priorities [2] and incremental resolving [8]. Recently, the design has been extended to allow multiple constraint solvers to cooperate to find a solution [7].

This design lends itself well to build interactive user interfaces for logic puzzles where the puzzle rules are expressed as constraints on the Morphic objects. In a standard imperative programming language, constraint solving and satisfaction is implemented explicitly. Using just Morphic in a standard imperative language, developers have to ensure that all event sources that might change the user interface resatisfy constraints or call an external constraint solver. In contrast, Babelsberg maintains constraints automatically, regardless of how the system was perturbed. This reduces the amount of knowledge the developer has to have about possible event sources for the Morphs. We argue that this is more in line with the encapsulation and abstraction desired in object-oriented applications.

An incomplete aspect of the existing Babelsberg design was that it only allowed constraints on objects and their parts, but did not operations on structures such as collections. In the context of logic puzzles the rules are usually defined on sets of objects (for example, Sudoku constraints are defined on rows, columns, and boxes.) We extended the Babelsberg design to support operations on collections of objects.

The contributions of this work are:

- We describe an implementation of the Babelsberg design in Squeak/Smalltalk.
- We describe extensions to Babelsberg that let the programmer conveniently specify constraints on collections.
- We present a technique for Morphic applications to interact with constraints, using as a running example an interactive Sudoku application

2. OBJECT CONSTRAINT PROGRAMMING IN SQUEAK

This section describes how constraints are expressed in our Squeak implementation of Babelsberg, called Babelsberg/S. For our examples, we use the rules of a Sudoku puzzle.

```
constraint := [(sudoku at: 1 at: 1) between: 1 and: 9]
                    alwaysSolveWith: solver.
```

Listing 1: Defining the domain of a Sudoku cell

Listing 1 shows the constraint for defining the domain of one Sudoku cell. In general, a constraint in Babelsberg/S is specified as a block that evaluates to a boolean — if the block evaluates to `true`, the constraint is satisfied. The constraint is created by sending the message `alwaysSolveWith:` to the block. The argument should be an instance of `ConstraintSolver`. It is also possible to solve the constraint with a default constraint solver, which is global inside the Squeak image, by sending `alwaysTrue`. As mentioned in Section 1, the constraint is defined by using object-oriented method definitions in Squeak rather than a DSL. The variable `sudoku` in Listing 1 represents the grid of cells in the interactive application and the method `between:and:` is a predefined predicate on Squeak numbers that just checks whether the receiver's value is between the upper- and lower-bound arguments.

Constraint Construction in Babelsberg/S.

To construct the constraint, the constraint block is executed in a different execution mode called constraint construction mode, which uses symbolic execution [3, 11] to create constraint expressions from the code. The block is only evaluated in constraint construction mode when either `alwaysTrue` or `alwaysSolveWith:` are sent to it, otherwise it is just an ordinary Squeak block.

After constraint construction has interpreted the block, the generated constraint expressions are added to a `Constraint` object, which is passed to the constraint solver. We explain the solving process in more detail in Section 3. If solving succeeds, the method `alwaysSolveWith:` returns

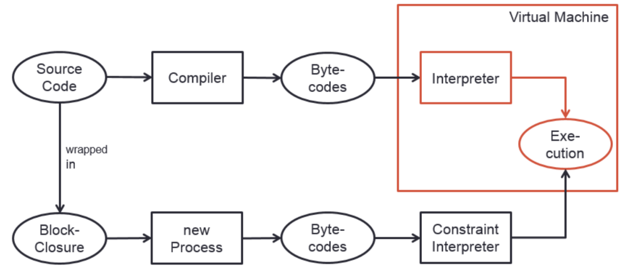

Figure 1: The architecture of Babelsberg/S constraint construction mode

the newly created constraint object. This object can then be used for reflection (e.g., to inspect which variables participate in the constraint) as well as to dynamically disable and re-enable the constraint. If solving fails, an exception is raised, which must be handled by the programmer. In that case, the constraint is not added and the system remains unperturbed.

Squeak/Smalltalk includes an in-image Smalltalk interpreter that we instrumented to implement constraint construction mode. The resulting architecture is shown in Figure 1. Squeak stack frames can be reified into instances of subclasses of the `ContextPart` class. These provide methods to interpret each bytecode. This facility is used by the Squeak debugger. Babelsberg/S uses the instrumented interpreter to evaluate the constraint block. The `alwaysTrue` method creates a new Process (a Smalltalk greenthread) that is interpreted stepwise using the interface of the `ContextPart` objects. Where interpretation in constraint construction mode deviates from normal Smalltalk semantics, we use ContextS [9] to instrument methods whose behavior needs to change inside a constraint construction mode layer.

Consider the above constraint: the block `[(sudoku at: 1 at: 1) between: 1 and: 9]` is compiled into bytecode. A new Squeak process is created (but not scheduled) by sending the method `newProcess` to it. The process has a stack with exactly one frame (a `ContextPart` object.) That frame's program counter is set to 0 and it contains the bytecode for the constraint block. The Babelsberg/S interpreter then steps through this frame by interpreting the bytecodes one by one, including doing method lookup and creating new frames as needed. An important consequence of this is that a variable binding that is used as receiver in a constraint block cannot be allowed to change, because then the lookup, and thus the constructed constraint, might be invalid. Thus, for Listing 1, the solver cannot simply find a collection that already satisfies the constraint and change the binding of the `sudoku` variable. Instead, it has to change the contents of the Sudoku to satisfy the constraint. This restriction does not apply to bindings that were created during constraint construction, such as return values of methods – so the solver can (and will) change what the method `at:at:` returns when sent to `sudoku`.

The modified interpreter creates `ConstraintVariable` objects for instance variables that are accessed through accessor methods. All methods are then called on these `ConstraintVariable` objects. Operator methods such as `+`, `-`, or `<=` construct constraint expressions instead of evaluating directly. Other methods that the solver does not directly support are partially evaluated to break them down into the primitive operations. In the case of `between:and:`, for example, the constraint constructed from partially evaluating the method would be equivalent to specifying `n >= lower and: [n <= upper]` directly. By re-using existing methods, Babelsberg/S supports the object-oriented abstractions that already exist in the system. This is equivalent to the Babelsberg implementations in Ruby and JavaScript [6].

Additionally, the interpreter creates instance-specific method wrappers to intercept access to these variables. The wrappers delegate read and write access to the corresponding `ConstraintVariable`, which calls the solver as needed to keep the constraints satisfied and returns the value of the variable from the solver's solution.

In contrast to JavaScript or Ruby, Squeak/Smalltalk does not allow instance-specific behavior directly. All methods and instance variables are declared on the class. However, wrapping accessors on the class of any encountered object would cause all instances of that class in the system to go through our wrapper, which imposes considerable performance overhead. To wrap only the encountered instances, we create anonymous subclasses of their class, and use Smalltalks `become:` facility to change the class of the object to the anonymous subclass. We then install our wrappers only on this instance-specific subclass.

This solution to instance-specific behavior means that there is no run-time overhead when using objects that have no constraints on them. Constrained objects are easily discovered through Smalltalk's meta-programming interface, because their class has no name and only wraps the accessors encountered in the constraint. We encountered methods in the core system that check for the class of its arguments not using the `isKindOf:` method (which works correctly for instances of subclasses), but by directly comparing the class pointer. Although one might consider this as a bug in the method, we are working on a solution to instance-specific behavior that is completely transparent to these common uses of meta-programming.

Constraints on Collections.

The existing Babelsberg design does not support constraints on collections directly; rather, it was proposed to use a specialized solver for collections [6]. To model an entire Sudoku puzzle, we need to assert the constraint given in Listing 1 for each cell. With the existing Babelsberg design, this would either require a solver for collections that supports domains for numbers, or alternatively, loop over the cells imperatively (Listing 2.)

```
(1 to: sudoku size) do: [:index |
   [(sudoku at: index) between: 1 and: 9]
       alwaysSolveWith: solver].
```

Listing 2: Defining the domain of all Sudoku cells with a loop

This code is incorrect if the size of the puzzle can change, because new elements will not have constraints on them. Additionally, if there is a method that hides the loop (e.g. `allCellsDo:`), it might not always be clear for developers if a method can be used in a constraint.

Babelsberg/S contributes to the development of more consistent and human-readable constraints by supporting the

`anySatisfy:`	$\exists x \in \text{array} : f(x)$
`noneSatisfy:`	$\forall x \in \text{array} : \neg f(x)$
`allSatisfy:`	$\forall x \in \text{array} : f(x)$

Table 1: Mapping from collection predicates to declarative representation

collection application programming interface (API) directly in constraints, rather than requiring a specialized solver for arrays. As a result, the domain constraint of a Sudoku puzzle can be expressed through sending `allSatisfy:` to `sudoku` (Listing 3.)

```
[sudoku allSatisfy: [:cell | cell between: 1 and: 9]]
   alwaysSolveWith: solver.
```

Listing 3: Defining the domain of all Sudoku cells with the Collection API

The extension to support collections directly in Babelsberg/S leverages the fact that Smalltalk comes with only one fixed-size pointer array type, upon which the Smalltalk collections library builds. This type provides three methods implemented in primitives for all low-level access: `at:`, `at:put:`, and `replaceFrom:to:with:startingAt:`.

Babelsberg/S subclasses the basic `Array` class and overrides the three low-level access methods to intercept any modifications to the array. In addition, it overrides the `copyFrom:to:` method, which is regularly used in Squeak to access sub-sequences of an array.

In constraint construction mode, any array that is visited in the dynamic extent of the execution is transparently replaced by the Babelsberg/S subclass. Besides the overridden methods, this subclass is a completely transparent proxy. Each element in the array that participates in the constraint is wrapped in a `ConstraintVariable`.

The predicates of the collection API are straightforward to support. The predicates `anySatisfy:`, `noneSatisfy:`, and `allSatisfy:` are mapped as per Table 1. Note that for the first relation, a disjunction over all elements must be created. For solvers that do not support disjunctions, Babelsberg/S forces the first element to satisfy the block. This prevents the system from finding solutions in many cases. To find additional solutions with solvers without disjunctions requires backtracking in the case of unsatisfiable constraints. This is not implemented yet, but is left for future work.

In general, any predicate method available on collections can be used in constraints. For example, additional predicate methods such as `allDifferent` are mapped to pair-wise inequalities by simply interpreting their implementation in constraint construction mode. Predicate methods are useful to ensure properties on all elements of a collection, for example, that they all be between 1 and 9 for a Sudoku.

Other methods that are useful in constraints reduce all elements of a collection and then express properties over those reductions. Reduction methods including `sum` and `count:`, which sum the elements or count the number of elements which satisfy a particular condition, are represented as linear expressions. The constraints created with these methods are reconstructed when the elements in the array change, but since the size of arrays is fixed, the length of the linear expressions is bounded. Such expressions are useful to state constraints on a collection as a whole, rather than on each of its elements. We have used this, for example, in our implementation of the Outside-Sum-Sudoku. Here, all elements in a collection must sum to the number outside the Sudoku. When one element changes, the others must change, too, to ensure the total sum does not.

We have found few use-cases for the most general collection-methods `do:`, `collect:`, and `select:` that could not be expressed using more specific methods. These methods create new collections from existing ones. What the developer means when using them and how to translate that meaning to the solver is less clear in the general case, and we do not support them for now. We have found that uses of `select:`, `collect:`, and `detect:` in predicate expressions can usually be replaced by the direct predicate methods. We have found that the iteration method `do:` is usually just used to express constraints on each element, and can usually be pulled outside of the constraint block. We might lift this restriction in the future if we find a significant number of uses of these methods in constraints that are much more expressive than their direct predicate counterparts. Until then, and for simplicity in the implementation, we do not allow these methods in constraints.

3. MAINTAINING CONSTRAINTS AUTOMATICALLY

Once asserted, constraints need to continue to be satisfied until they are disabled, all objects they apply to are garbage collected, or the program stops. To ensure this, the Babelsberg design follows the *perturbation* model established by the Kaleidoscope constraint-imperative language [12]. This model is similar to reactive systems in that changes to one part of the system propagate to other parts. In reactive systems, these changes are made by sampling a continuous process or through discrete events. In Babelsberg, the changes are the concrete event of assigning a new value to a variable that participates in a constraint. These changes then potentially propagate to other variables to keep constraints satisfied.

Each variable that participates in a constraint implicitly reacts to programmatic changes to its value by calling one or more solvers to re-satisfy the variable's constraints. Our wrappers around accessors intercept changes to variables that were used in a constraint and call `suggestValue:` on their associated `ConstraintVariable`. This adds a temporary equality constraint for the new value to the underlying constraint solver. The solver tries to solve all constraints. If the constraints are satisfiable, the new value is assigned. As a side-effect, other variables might change to satisfy constraints. If the solver cannot find a solution, a runtime error is generated and the new value is ignored.

In our Sudoku example, if a new value is assigned to a cell, the Sudoku constraints are solved in the background. If the solver finds a solution, the cell changes its value and the rest of the puzzle is adjusted to keep the Sudoku solveable. That is possible because the Sudoku application interacts with the underlying constraints.

Babelsberg/S can accommodate a variety of constraint solvers. Currently, it supports the Squeak implementation of Cassowary [1], and Z3 [4] through an IPC interface.

4. EVALUATION

The constraints in Sudoku are easy to state, but not always easy to satisfy. A correct solution must assign each cell

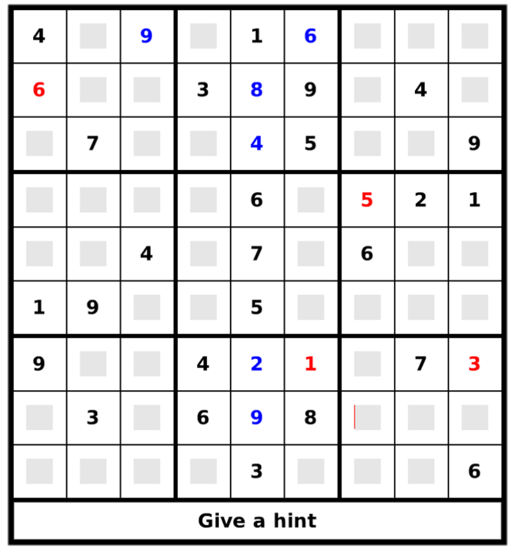

Figure 2: The Morphic UI of the Sudoku puzzle

a number between 1 and 9 inclusively, while at the same time ensuring the no number occurs twice in a row, a column, or a block of 3 by 3 cells. We argue that logic puzzles such as Sudoku are good examples for interactive constraint applications. The user interface (written in Morphic) is shown in Figure 2. Some numbers (in black) are given initially. Each cell is a standard Morphic text box, which allows the user to input a single character (in blue). The system can also generate hints (printed in red).

Listing 4 shows the constraints necessary to solve a Sudoku puzzle. These constraints use the Z3 constraint solver. Line 1 ensures that the user cannot change the numbers that were given initially. In some solvers, such as Cassowary, stay constraints can be used to express that the solver may not change a given variable, or to only change it if the constraints cannot be satisfied otherwise. Stay constraints are currently not supported in Z3, but will be in future versions. Currently, the method `addConstraintsForAllGivenNumbers` iterates over cells and creates a constraint that each cell that already has a value is always equal to just that value. Lines 3–4 assert the constraint that all cells must contain numbers between 1 and 9. Finally, lines 6–10 ensure that no row, column, or 3×3 box of cells can have duplicate numbers.

Note that the Squeak collection API does not contain a method `allDifferent`. Babelsberg/S adds this predicate for convenience. It is a normal object-oriented method in the `Collection` class that iterates over all elements in the collection and tests them for pairwise inequality. In ordinary code, this is just a test – the constraint interpreter, however, creates an inequality constraint expression for each comparison, exploding the allDifferent method into multiple constraints that the solver can understand. This means also, that subclasses can override the method and any different behavior will be reflected in the created constraints.

Note also that the normal accessor methods for rows and columns from Squeak `Matrix` objects are used, too. The Sudoku grid is just a subclass of `Matrix` that, besides a method to assert constraints on the given numbers, adds the `atBox:` accessor method to access each of the 9 boxes of size 3×3.

```
sudoku addConstraintsForAllGivenNumbers.

[sudoku allSatisfy: [:cell | cell between: 1 and: 9]]
    alwaysSolveWith: solver.

(1 to: sudoku rowCount) do: [:index |
    [(sudoku atRow: index) allDifferent &
    (sudoku atColumn: index) allDifferent &
    (sudoku atBox: index) allDifferent]
        alwaysSolveWith: solver].
```

Listing 4: All Constraints of a Sudoku Puzzle

As can be seen from Listing 4, the amount of code necessary for specifying all properties of a Sudoku puzzle is very small. With these, a solver can solve an arbitrary given Sudoku puzzle. The constraints are completely decoupled from the specific Sudoku puzzles and their given numbers.

In Babelsberg, constraints can be constructed, enabled and disabled at run-time, and, because they work correctly with method polymorphism, it is possible to subclass a logic puzzle to construct another by adding or removing constraints only. As an example, we have created Sudoku puzzle subclasses for *Diagonal Sudokus* and *Outside-Sum Sudokus*. In the former, the numbers of the two main diagonals have to be all different, and in the latter, the first three numbers in a row or a column must add up to a specific sum.

For a Diagonal Sudoku, provided there are accessor methods for the two diagonals, the method to create constraints is shown in Listing 5.

```
super createConstraints. "from normal Sudoku"
[(self diagonalFromTopLeft allDifferent)
  and: [self diagonalFromTopRight allDifferent]]
    alwaysSolveWith: solver.
```

Listing 5: The Diagonal Sudoku

With object-constraint programming (OCP), it does not matter in which way a constraint variable or a constraint changes. The constraint satisfaction automatically works on each disturbance of the system. Currently, the values of cells only change when the user enters a new value into the morph that represents a cell. If that value is not a number between 1 and 9, or the Sudoku cannot be solved by adding this value, the solver rejects the input. However, the constraints encode no source for the change, so it does not matter if the change actually occurred through keyboard input. The Sudoku could also be calculated entirely by the computer, or the game could allow remote users to send values over the network. The constraints thus provide flexibility, because the developer does not need to know all events that might change the puzzle.

5. CONCLUSIONS

We have argued that OCP facilitates reactive systems in which dependencies between objects can be declared as constraints. It modularizes the relationship between objects and decouples constraint satisfaction from the application. Constraints can be dynamically added and removed, and are maintained automatically. This makes them useful for writing interactive applications. As an example, we implemented applications for specifying and solving different variants of Sudoku with constraints with a graphical user interface. The user can change the values of the constraint variables interactively without breaking the properties of the Sudoku. The application reacts on the user input by resolving the underlying constraints.

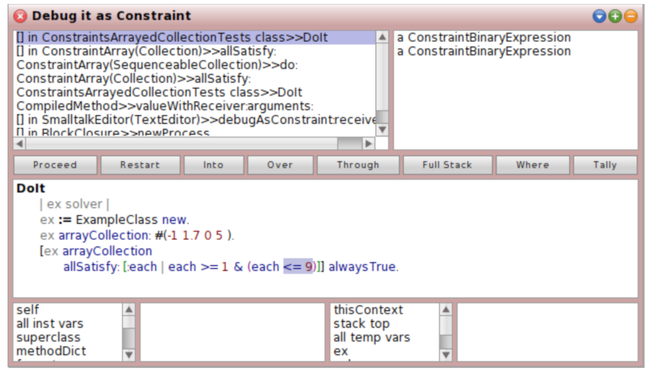

Figure 3: Squeak debugger with constraints

There are two major directions for future work. Regarding the implementation, we plan to implement an alternative solution to provide instance-specific wrappers. This will improve the compatibility of constrained objects with existing Smalltalk code. We also plan to support more features of the Babelsberg design as found in its JavaScript and Ruby implementations, such as incremental resolving, local propagation, and identity constraints.

Furthermore, we plan to leverage the Smalltalk metaprogramming facilities to explore how to aid developers in debugging and understanding constraints. If incorrect constraints are generated, why? If the solver cannot find a solution or is slow, what can be done? These are still open questions for Babelsberg, because constraints cannot be easily debugged. In Babelsberg/S, because we are re-using the Squeak debugger's bytecode interpreter, we have extended the debugger to support stepping into constraints (see Figure 3). When stepping through code with constraints, the debugger shows which constraints are created and when the solver is invoked. If developers suspect that constraints were created incorrectly, they can remove the constraint, change the code, and restart the method as appropriate. We plan to extend this prototype into a debugger that is useful to answer different questions that arise when developing with constraints.

Despite these avenues for future work, we think that Babelsberg/S is already a useful implementation of object-constraint programming and we plan to include it in a future release of the R/Squeak distribution, a Squeak distribution that includes research projects considered useful for general purpose development[1].

6. REFERENCES

[1] G. J. Badros, A. Borning, and P. J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, Dec. 2001.

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

[3] L. A. Clarke. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, 2(3):215–222, 1976.

[1] https://www.hpi.uni-potsdam.de/swa/trac/SqueakCommunityProjects

[4] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[5] Enthought Inc. Enaml 0.6.3 documentation. `http://docs.enthought.com/enaml/`, Feb. 2014.

[6] T. Felgentreff, A. Borning, and R. Hirschfeld. Specifying and solving constraints on object behavior. *Journal of Object Technology*, 13(4):1–38, Aug. 2014.

[7] T. Felgentreff, A. Borning, R. Hirschfeld, J. Lincke, Y. Ohshima, B. Freudenberg, and R. Krahn. Babelsberg/JS. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 411–436. Springer, July 2014.

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

[9] R. Hirschfeld, P. Costanza, and M. Haupt. An introduction to context-oriented programming with ContextS. In *Generative and Transformational Techniques in Software Engineering II*, pages 396–407. Springer, July 2008.

[10] I. L. Ist, I. Lynce, and J. Ouaknine. Sudoku as a sat problem. In *Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics, AIMATH 2006, Fort Lauderdale*, pages 1–9. Springer, Jan. 2006.

[11] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[12] G. Lopez, B. Freeman-Benson, and A. Borning. Kaleidoscope: A constraint imperative programming language. In *Constraint Programming*, volume 131, pages 313–329. Springer, 1994. NATO Advanced Science Institute Series, Series F: Computer and System Sciences.

[13] J. Maloney. *Morphic: The Self User Interface Framework*. Self, 4th edition, 1995.

[14] J. Maloney. *An introduction to morphic: The squeak user interface framework*. Imagineering, Walt Disney, 2001. In *Squeak: OpenPersonal Computing and Multimedia*.

[15] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *33rd International Conference on Software Engineering (ICSE)*, pages 511–520. IEEE, May 2011.

[16] E. Sadun. *iOS Auto Layout Demystified*. Addison-Wesley, Oct. 2013.

[17] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web browser as an application platform: The lively kernel experience. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, Apr. 2008.