Solving Interactive Logic Puzzles With Object-Constraints

An Experience Report Using Babelsberg/S for Squeak/Smalltalk

Maria Graber¹ Tim Felgentreff² Robert Hirschfeld²
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.

Listing 1: Defining the domain of a Sudoku cell

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

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 Constraint-Solver. 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 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 green-thread) 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 evalu-
ing 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 a >= lower and: [a <= 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 Smalltalk’s become: 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).

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

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

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 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).

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

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

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

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.

6. REFERENCES


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

[5] Enthought Inc. Enaml 0.6.3 documentation. 

Specifying and solving constraints on object behavior. 

[7] T. Felgentreff, A. Borning, R. Hirschfeld, J. Lincke, 
Y. Ohshima, B. Freudenberg, and R. Krahn.


Unifying execution of imperative and declarative code. 
