Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Babelsberg: Specifying and Solving Constraints on Object Behavior

Tim Felgentreff[a,c] Alan Borning[b,c] Robert Hirschfeld[a,c]

a. Hasso Plattner Institute, University of Potsdam

b. University of Washington

c. Communications Design Group, SAP Labs

Abstract Constraints allow developers to specify desired properties of systems in a number of domains, and have those properties be maintained automatically. This results in compact, declarative code, avoiding scattered code to check and imperatively re-satisfy invariants. Despite these advantages, constraint programming is not yet widespread, with imperative programming still the norm.

There is a long history of research on integrating constraint programming with the imperative paradigm. However, this integration typically does not unify the constructs for encapsulation and abstraction from both paradigms. This impedes re-use of modules, as client code written in one paradigm can only use modules written to support that paradigm. Modules require redundant definitions if they are to be used in both paradigms.

We present a language – Babelsberg – that unifies the constructs for encapsulation and abstraction by using only object-oriented method definitions for both declarative and imperative code. Our prototype – Babelsberg/R – is an extension to Ruby, and continues to support Ruby's object-oriented semantics. It allows programmers to add constraints to existing Ruby programs in incremental steps by placing them on the results of normal object-oriented message sends. It is implemented by modifying a state-of-the-art Ruby virtual machine. The performance of Ruby code without constraints is only modestly impacted, with typically less than 10% overhead compared with the unmodified virtual machine. Furthermore, our architecture for adding multiple constraint solvers allows Babelsberg to deal with constraints in a variety of domains.

We argue that our approach provides a useful step toward making constraint solving a useful tool for object-oriented programmers. We also provide example applications, written in our Ruby-based implementation, which use constraints in a variety of application domains, including interactive graphics, circuit simulations, data streaming with both hard and soft constraints on performance, and configuration file management.

Keywords Constraints, Object Constraint Programming

1 Introduction

Constraints and constraint programming occur in a variety of application domains within computer science and related fields, including graphics, CAD/CAM, planning and operations research, artificial intelligence, user interface toolkits, and programming languages. A *constraint* is simply a relation that should hold. Some examples of constraints from a variety of application domains are: that there be a minimum of 10 pixels horizontal space between two buttons on a screen, that a resistor in an electrical circuit simulation obey Ohm's Law, that a maximum of 10 parts per hour can be produced by a machine in a factory, that all of the digits in a row or column or subregion of a Sudoku puzzle must be different, that the height of a bar in a bar chart correspond to a value produced by a simulation, or that a streamed video play smoothly in the presence of changing CPU and network load. Constraints are declarative: they specify *what* should be the case rather than *how* to achieve it. They thus provide flexibility in the way they can be employed, since a single constraint can typically be used in multiple ways.

A number of features are desirable in constraint programming to solve a useful set of problems. Unfortunately it is in general much easier to state constraint problems than to solve them — some problems are extremely difficult, and others are undecidable. On the other hand, for some restricted but very useful classes of constraints, quite efficient solvers are available. Therefore, practical solvers impose restrictions on the kinds of constraints they can solve, for example, by supporting only floats and not integers or requiring numeric equalities and inequalities to be linear. For example, the Cassowary solver [BBS01] can solve multi-way constraints on floats; Z3 [DMB08] can do the same for reals, integers, and booleans; Kodkod [TJ07], Z3, and Ilog [Pug94, IBM14] can enumerate solutions; and DeltaBlue [FBM89] can solve multi-way constraints using local propagation. An even more restricted approach is to only consider one-way constraints that can compute an output value given new inputs, but not vice versa (e.g., [HS96, MGD+90]).

Thus, in general a variety of solvers are needed, depending on the type domain and particular characteristics of the constraints. It is also useful to have these solvers interoperate to find a solution to a particular problem. Furthermore, in some application domains, it is important to support soft as well as hard constraints, that is, constraints that should be satisfied if possible as well as constraints that must be satisfied. Finally, in the presence of frequent changes, incremental solvers are required that can efficiently re-solve a set of constraints as input values are changed or constraints added and deleted. Interactive graphical systems provide examples of all of these: they include constraints on complex objects such as points and rectangles as well as floats and integers, soft constraints to express preferences regarding layout, and require incremental solvers to efficiently re-solve constraints repeatedly, for example when a part of the figure is moved.

In this work, we focus on constraints over objects and primitive types such as floats, integers, and booleans (e.g., for interactive graphical applications) that can describe system invariants, for example to write systems that adhere to a specification, as well as highly dynamic constraints that can be created, added, modified, and removed at run-time. In contrast, this work does not support constraints for searching or enumerating multiple solutions to a constraint model.

There are two main alternatives for incorporating constraints into imperative programs. The most common way is via a constraint solver library. This has the advantage that no changes are required to the underlying programming language, but the disadvantage that the solver needs to be invoked explicitly at appropriate times during the execution. If it is not, the programmer may inadvertently ignore or bypass the constraints. An alternate technique is to support constraints directly in the underlying programming language. This addresses the problem of inadvertently bypassing the constraint system, because the programming language ensures that the solver is invoked whenever neccessary. This approach also typically provides a more convenient syntax for writing constraints, either using a separate domain specific language (DSL) in conjunction with the host language (with the drawback of having to learn essentially two programming languages), or (as in the work reported here) by allowing constraints to be written in the host language itself. In both cases there is the cost of having to support a new or extended programming language.

Prior approaches of the latter kind typically provide a unified runtime, a semantics for the interactions between declarative and imperative code, and a linguistic symbiosis. As an extension of these, this work additionally unifies the programming constructs for encapsulation and abstraction by using only object-oriented methods, classes, and inheritance for defining behavior and structure in both the imperative and declarative paradigm.

We present our design for integrating constraints with a dynamic, object-oriented language that preserves a familiar imperative programming model, called Babelsberg, and describe an implementation of that design as an extension to Ruby [FM08], called Babelsberg/R. Key contributions for Babelsberg with respect to language design are as follows.

- The semantic model for Babelsberg is a direct extension of an object-oriented model with dynamic typing, object encapsulation, classes and instances or prototypes with methods, and message sends. It supports placing constraints on the results of message sends rather than just on object attributes — thus, we argue, being more compatible with object-oriented encapsulation and abstraction than prior approaches in constraint imperative programming (CIP). The only restrictions are: a) an expression that is used as a constraint must evaluate to a boolean (the constraint is that it evaluate to true); b) the expression should return the same result on repeated evaluation (so that, for example, a random number generator would not qualify); and c) variables used in the expression must be used in single assignment fashion.
- The syntax of Babelsberg is a strict superset of that of the base language — with only one minor extension to allow a question-mark as method name — making it easy for the programmer to write and read constraints and object-oriented code.
- Babelsberg includes meta-level facilities that allow libraries to construct soft as well as hard constraints, and constraints that support incremental solving.

Additional contributions with respect to implementation techniques are as follows.

- We present a technique for implementing object constraint languages, which uses a primitive to switch the interpreter between imperative evaluation, constraint construction, and constraint solving modes. In imperative evaluation mode, the interpreter operates in the standard fashion, except that LOAD and STORE operations on variables check for constraints, and if present, obtain the variable's value from the constraint solver or send the new value to the solver (which may trigger a cascade of other changes to satisfy the constraints). In constraint construction mode, the expression that defines the constraint is evaluated, not for its value, but rather to build up a network of primitive constraints that represent the constraint being added. The interpreter keeps track of dependencies in the process, so that, as needed, the solver can be activated or the code to construct the constraint can be re-evaluated.

- Babelsberg provides an architecture that supports multiple constraint solvers, which makes it straightforward to add new solvers, and which does not privilege the solvers provided with the basic implementation (they are simply the solvers that are in the initial library).
- We describe a working prototype system, integrated with a state of the art Ruby virtual machine and just-in-time (jit) compiler. In the absence of constraints, the performance of a program written in the host language (Ruby) is only modestly impacted.

Babelsberg/R is a general-purpose object constraint language, and we have implemented a variety of example programs, including a video streaming example with both hard and soft constraints on performance, interactive graphical layout examples (which exercise soft constraints and incrementality), electrical circuit simulations, and solvers for puzzles such as Eight Queens and cryptarithmetic.

The rest of this paper is structured as follows. Section 2 presents an overview of related work in constraint programming. Section 3 provides the theoretical background for constraint priorities and incremental solving. In Section 4, we present the features of Babelsberg; how these features are implemented is described in Section 5. Section 6 presents our performance evaluation, applications written in Babelsberg/R, and a comparison of Babelsberg's features with related approaches. Section 7 describes future work and concludes. A companion technical report [FBH14a] provides additional details and sample Babelsberg programs in an appendix. (The body of the technical report is the same as an earlier draft of this article.)

2 Related Work

Consider a graphical rectangle implemented as a pair of points – origin and extent – with a method to test whether the origin is in the visible (i.e., positive) part of the coordinate system, and a method to calculate its area.

```
class Rectangle
  attr_accessor :origin, :extent

  def visible?
    origin.x >= 0 and origin.y >= 0
  end

  def area
    extent.x * extent.y
  end
end
```

Suppose this rectangle encompasses some information that we want to make sure remains on screen entirely. To that end, the area of the rectangle should never be less than 100 square pixels and its origin should remain visible. The constraints are that the `visible?` method always returns true, and the area method always returns a value ≥ 100. (Note that the `visible?` method contains two inequalities to form a conjunctive constraint – both conditions must be satisfied to satisfy the constraint. It would be straightforward to include two additional tests that `origin.x + extent.x <= DISPLAYWIDTH` and `origin.y + extent.y <= DISPLAYHEIGHT`; these are omitted for simplicity.)

For imperative languages, the usual approach to dealing with such dynamic constraints is to leave it entirely up to the programmer to ensure that they are satisfied — the constraints are either just in comments and documentation, or perhaps in the form of machine-checkable assertions. For the latter case, programmers typically write assertions to fail early if these constraints are unexpectedly not satisfied [RCN05].

Alternatively, in an object-oriented language like Ruby, we might ensure that these constraints are satisfied, for example using aspect-oriented programming [KLM+97] to intercept writes to either origin or extent (lines 9–10 in the following code) and execute a corrective action (lines 3–5).

```
1  class RectAspect < Aspect
2    def ensure_constraints(method, rect, status, *args)
3      rect.origin.x = 0 if rect.origin.x < 0
4      rect.origin.y = 0 if rect.origin.y < 0
5      rect.extent.x = 100.0 / rect.extent.y if rect.area < 100
6    end
7  end
8
9  RectAspect.new.wrap(Rectangle, :postAdvice, "extent=")
10 RectAspect.new.wrap(Rectangle, :postAdvice, "origin=")
```

However, the above code has a number of problems:

- The constraints are expressed in a form that makes them non-obvious — the branch conditions have to be carefully checked to ensure they catch all undesired states, and the reader has to infer the constraint from the conditions (for example, that the checks for non-negativeness express the visibility constraint.)
- All modifications that may invalidate the constraints have to be intercepted — if the advice is insufficient or not executed at the correct times, the constraints may be violated through parts of the execution.
- There are multiple possible solutions to the constraints, but which one should be selected is not explicit in the code. (For example, a rectangle with area 200 instead of 100 would also satisfy the minimum area constraint.) Without a declarative specification, it is nontrivial to decide whether this code will find the optimal solution. In the presence of competing soft constraints, i.e., multiobjective optimizations, finding a solution becomes even harder (as discussed in Section 3.)

2.1 Constraint Solver Libraries

To address the aforementioned problems, another approach is to use a library that provides one or more constraint solvers that allow us to express our constraints explicitly. There is a huge range of libraries that can be called directly from the imperative code. One way of classifying them is by the *type domain* of the constraints (for example, real numbers or finite domains or arbitrary objects); whether their solver can solve constraints in a general way or just select from a given set of functions (e.g., given $c = a + b$, can it solve for any of a, b, or c, or several variables simultaneously; or can it only find a value for c given a and b); and whether the solver supports particular features, such as incremental solving.

A few solvers of particular interest for the programming language community are Z3 [DMB08], a state-of-the-art SMT solver from Microsoft Research designed for theorem proving (e.g., for program verification), and kodkod [TJ07] for constraints over finite domains. Solvers for use in interactive graphics systems include Cassowary [BBS01], an incremental solver for linear equality and inequality constraints that supports soft constraints as well as hard ones, the Auckland Layout Editor [LW08], which includes support for a GUI builder using constraints, and earlier work on DeltaBlue [FBM89], which supports multi-way local propagation constraints and soft constraints. There is also a range of commercial solvers and applications, such as the CPLEX optimizer for mathematical programming [ILO93].

For example, the following code rewrites the previous solution that relied on state changing operations and branches, to use the Z3 library to solve our constraints (lines 6 and 9). Additional code is required to copy state between the solver and the rectangle object.

```
 1 class RectAspect < Aspect
 2   def ensure_constraints(method, rect, status, *args)
 3     ctx = Z3::Context.new
 4     ctx ≪ Z3::Variable.new("origin_x", rect.origin.x)
 5     ctx ≪ Z3::Variable.new("origin_y", rect.origin.y)
 6     ctx ≪ Z3::Constraint.new("origin_x >= 0 && origin_y >= 0")
 7     ctx ≪ Z3::Variable.new("extent_x", rect.extent.x)
 8     ctx ≪ Z3::Variable.new("extent_y", rect.extent.y)
 9     ctx ≪ Z3::Constraint.new("extent_x * extent_y >= 100")
10     ctx.solve
11     rect.extent.x = ctx["extent_x"]
12     rect.extent.y = ctx["extent_y"]
13   end
14 end
15
16 RectAspect.new.wrap(Rectangle, :postAdvice, "extent=")
17 RectAspect.new.wrap(Rectangle, :postAdvice, "origin=")
```

Using a solver allows programmers to express constraints about the system in terms of a solver-specific type domain (e.g., reals, booleans, uninterpreted function symbols) the solver understands. If the problem is expressible in a type domain for which a solver is available (as the above code is) the constraints can be written clearly.

2.2 Domain-specific Languages for Constraints

For specialized application domains, constraints are sometimes available via separate DSLs. For user interface layouts, DSLs describe relations between visible objects that can be automatically maintained by the runtime. Examples are CSS [LB97], the Mac OS X [Sad13] layout specification language (which uses Cassowary to solve the constraints), and the Python GUI framework *Enaml* [Ent14]. These allow programmers to express relations such as distances between objects or parent/child alignments. These constraints are automatically re-satisfied by the runtime when imperative code changes the user interface.

The following is an example of an *Enaml* specification for our problem:

```
enameldef Main(Window):
    Container:
        constraints = [
            # the rectangle area is called contents in enamel
            contents_top >= 0, contents_left >= 0,
            (contents_bottom - contents_top) *
              (contents_right - contents_left) >= 100
        ]
```

DSLs allow programmers to use constraints in specific application domains without having to write boilerplate code for triggering constraint solving. They have found widespread adoption and renewed interest recently, in particular through the Mac OS X layout system.

2.3 Dataflow Constraints and FRP

Some languages have built-in support for data flow, which allows programmers to express unidirectional constraints between objects and their parts. Examples of such systems are Scratch [RMMH+09], LivelyKernel/Webwerkstatt [LKI+12], and KScript [OLFK13].

The following uses LivelyKernel connections to observe changes to `origin` and `extent` in a Rectangle `rect`. On each change, the transformation function is executed with the current and the previous value and returns the new value for the field.

```
connect(rect, "origin", rect, "origin",
        function(origin, prevOrigin) {
            if (!this.isVisible()) return prevOrigin;
            else return origin;
        })
connect(rect, "extent", rect, "extent",
        function(extent, prevExtent) {
            if (this.area() < 100) return prevExtent;
            else return extent;
        })
```

Although these systems are not constraint solvers, programmers can use constraint solvers (in the hook function passed to `connect`) to calculate new values and some systems, like KScript, already integrate a constraint solver to use in the connection. These approaches provide one answer to the question of when to trigger constraint solving.

2.4 Integrating Constraints with a Programming Language

Another approach to supporting constraints — and the one adopted in the work reported here — is to integrate constraint support with a general purpose programming language. Again, there is a substantial body of prior work in this area.

One of the earliest examples for this approach is the Constraint Logic Programming scheme [JL87], which evolved from logic programming. One instance, CLP($\mathcal{R}$) [JMSY92], provides constraints over real numbers in Prolog. These languages are in the logic programming family, and in their standard form have no notion of state or state change. Another language of this kind is Concurrent Constraint Programming [SR93].

Such languages have significant advantages, such as a clean semantics, but they sacrifice the familiar capabilities and programming style of the more mainstream object-oriented paradigm. Our goal here is to support an object-oriented, imperative programming style that integrates constraints syntactically and semantically.

With these goals, Babelsberg follows the work by Freeman-Benson, Lopez, and Borning on CIP [FBB92b, LFBB94a, LFBB94b, LFBB94c] and the Kaleidoscope language. Systems related to Kaleidoscope include Siri [Hor92], Turtle [GH04], and Soul [DGJ04]. These languages provide constraints with syntactic integration into an object-oriented language and allow expressing constraints on all object-oriented values in their language. However, they require separate abstractions and method definitions for the declarative and imperative paradigm.

BackTalk [RP97] and Ilog Solver [Pug94, IBM14] are other systems that aim to integrate a rich set of constraint solvers with imperative languages, but without full syntactic integration or re-using of object-oriented method definitions. Rather, they provide a library for unified access to a wide range of solvers to be able to express various constraints, and methods to combine these solvers with boolean operators or propagation. More recently, Kaplan [KKS12] syntactically integrated constraints with Scala, but again, does not allow programmers to re-use object-oriented method definitions to express constraints, and requires objects to be wrapped as *logical variables* to use them in constraints. All three systems allow the programmer to enumerate possible solutions to constraints, a goal CIP languages and the work presented here do not share.

There is also a body of work that uses constraints in other ways in general-purpose programming languages. For example, in Plan B, Samimi, Aung, and Millstein [SAM10] use specifications as “reliable alternatives” to implementations, so that if an assertion fails, the system can use the specification as input to a constraint solver and continue execution. Similarly, Demsky and Rinard [DR06] use constraint solvers to correct a faulty program state automatically and continue running. (Thus, by replacing assertions with constraints, undesirable program states can be corrected by the runtime.)

Constraint Imperative Programming in Kaleidoscope

Because the current work shares many of its motivations with Kaleidoscope, it also shares important design aspects that were developed over the iterations of the Kaleidoscope language. At the same time, it embodies some significant improvements over that earlier work, in particular, by providing a simpler semantic model that uses ordinary methods and messages, rather than requiring a separate concept of constraint constructors.

Abstractions Kaleidoscope supported Smalltalk-like classes and instances, and in addition, integrated constraints with the language itself. This integration included built-in constraints over primitive objects (such as floats) and constraints over user-defined objects, which were provided by *constraint constructors*. For example, the `+` constraint for Points could be defined using a constraint constructor `a+b=c` that expanded into constraints on the `x` and `y` instance variables of the three points `a`, `b`, and `c`. Separately, the language also provided object-oriented (OO) methods. Both constraint constructors and OO methods were selected using multi-method semantics. This accommodated, for example, the case of a constraint constructor call `a+b=c` in which `b` and `c` were known and `a` was unknown.

Our rectangle example can be expressed using Kaleidoscope as follows:

```
class Rectangle
  constructor area = (n: Integer)
    always: extent.x * extent.y = n
  end

  constructor visible?
    always: origin.x >= 0
    always: origin.y >= 0
  end
end

rect = Rectangle.new
always: rect.area = 100
always: rect.visible?
```

With CIP as in Kaleidoscope, programmers can express constraints over ordinary objects if the required *constraint constructors* are added to their classes in addition to ordinary methods.

In contrast to Kaleidoscope, BABELSBERG provides a simpler semantic model. BABELSBERG uses the object-oriented model with ordinary methods using OO message dispatch, rather than special constraint constructors and multi-methods. The implementation includes an integration with a state of the art virtual machine and JIT, so that in the absence of constraints, the performance of a program written in the host language is only modestly impacted.

Mutable State This work's treatment of mutable state and time in BABELSBERG is very similar to the later incarnations of Kaleidoscope. The first version of Kaleidoscope, Kaleidoscope'90 [FBB92a], used a *refinement* model, in which variables held a stream of values, related to each other by constraints. Variables typically had a low-priority *stay* constraint so that they retained their value over time, e.g., $x_t = x_{t-1}?$. (The question mark is a read-only annotation: the constraint solver wasn't allowed to change the past to satisfy constraints on the present.) There were facilities to access both the current and previous states of a variable. Object identity was only an implementation issue in Kaleidoscope'90, and not semantically significant. Later versions of the language (e.g., Kaleidoscope'93) [LFBB94a, LFBB94b] switched to a *perturbation* model, in which destructive assignment can change the state of objects (perhaps making previously satisfied constraints unsatisfied), and the system perturbs or adjusts values to reach a new state that best satisfies the constraints. Instead of streams of values, a variable in Kaleidoscope'93 referred to a single object, as in a more conventional language. Kaleidoscope'93 also made object identity a part of the language semantics, including support for identity constraints as well as equality constraints.

3 Hard and Soft Constraints

As noted in the introduction, in some application domains, it is important to support soft as well as hard constraints, that is, constraints that should be satisfied if possible and constraints that must be satisfied. In BABELSBERG we use the semantics for hard and soft constraints presented in reference [BFBW92], which we briefly review here.

Informally, we consider collections of constraints, each labeled with a *priority*. There is a distinguished priority *required*, which denotes constraints that must be satisfied in any solution. There are an arbitrary number of other priorities for soft constraints. The priorities are totally ordered, with higher-priority constraints satisfied in preference to lower-priority ones. While both the theory and the constraint solvers that we use can accommodate an arbitrary number of priorities, in practice programmers tend to employ only a small number of priorities in stylized ways [BBS01]. In the examples in this section, we will use the priorities `required`, `high`, `medium`, and `low`.

For example, consider the following constraints over the reals:

$$\begin{array}{rl} \text{required} & x + y = 10 \\ \text{high} & x = 8 \\ \text{low} & x = 0 \\ \text{low} & y = 0 \end{array}$$

There is a single solution that best satisfies the constraints, namely $\{x \mapsto 8, y \mapsto 2\}$, since this satisfies both the required and high-priority constraints (and no other solution does so).

Soft constraints can have an associated *error function* that returns 0 iff the constraint is satisfied. A simple error function just returns 0 if the constraint is satisfied, and 1 if it is not; but for reals, we often use a *metric* function whose value increases smoothly the further the variable's value is from the desired one. For example, the metric error for a constraint $x = y$ is simply $|x - y|$, while the error for $x \leq y$ is $x - y$ if x is greater than y, and otherwise 0. For example, consider the following constraints:

$$\begin{array}{rl} \text{required} & 10 \leq x \leq 20 \\ \text{high} & 15 \leq x \\ \text{low} & x = 5 \end{array}$$

Using a metric error function, the solution is $\{x \mapsto 15\}$, since this completely satisfies the required and high-priority constraints, and minimizes the error for the low-priority one.

We might have two constraints with the same priority that cannot be satisfied simultaneously:

$$\begin{array}{rl} \text{required} & x + y = 10 \\ \text{medium} & x = 0 \\ \text{medium} & y = 0 \end{array}$$

Reference [BFBW92] describes a number of different *comparators* for specifying the desired solution. Two comparators of particular interest here are *locally-predicate-better* and *weighted-sum-better*. Locally-predicate-better finds a Pareto-optimal solution, that is, a solution such that any other solution would cause a currently-satisfied constraint with priority p to be unsatisfied, and further, that no higher-priority constraints would become satisfied. Weighted-sum-better finds solutions that minimize the weighted sum of the errors for the highest-priority constraints; if there are multiple solutions considering just the required and highest-priority constraints, we consider the next highest priority, and so on. In the above example, there are two locally-predicate-better solutions, $\{x \mapsto 0, y \mapsto 10\}$ and $\{x \mapsto 10, y \mapsto 0\}$. If the weights on the two medium-priority constraints are the same, there are an infinite number of weighted-sum-better solutions, namely all $x \in [0, 10], y \in [0, 10]$ such that $x + y = 10$. However, if we make the weight on say the $x = 0$ slightly higher, we get a single solution, namely $\{x \mapsto 0, y \mapsto 10\}$.

In Babelsberg, we are interested in solvers that find *a* solution to a collection of constraints, so if there are multiple solutions, the solver is permitted to select one arbitrarily. (This is in contrast to logic programming and constraint logic programming languages, which give access to multiple or all solutions, perhaps found by backtracking.)

Formally, a constraint is a relation over some domain $\mathcal{D}$. The domain $\mathcal{D}$ determines the constraint predicate symbols $\Pi_{\mathcal{D}}$ of the language, so that a constraint is an expression of the form $p(t_1, \ldots, t_n)$ where p is an n-ary symbol in $\Pi_{\mathcal{D}}$ and each t_i is a term.

A *labeled constraint* is a constraint labeled with a priority, written pc, where p is a priority and c is a constraint. For clarity in writing labeled constraints, we give symbolic names to the different priorities. We then map each of these names onto the integers $0 \ldots n$, where n is the number of non-required priorities. Priority 0, with the symbolic name `required`, is always reserved for required constraints.

A constraint system is a multiset H of labeled constraints. Let H_0 denote the required constraints in H, with their labels removed. In the same way, we define the sets $H_1, H_2, \ldots, H_n$ for levels $1, 2, \ldots, n$. We also define $H_k = \emptyset$ for $k > n$.

A *solution* to a set of labeled constraints H is a valuation for the free variables in H, i.e., a function that maps the free variables in H to elements in the domain $\mathcal{D}$. We wish to define the set S of all solutions to H. Clearly, each valuation in S must be such that, after it is applied, all the required constraints hold. In addition, we desire each valuation in S to be such that it satisfies the non-required constraints as well as possible, respecting their relative strengths. To formalize this desire, we first define the set S_0 of valuations such that all the H_0 constraints hold. Then, using S_0, we define the desired set S by eliminating all potential valuations that are worse than some other potential valuation using the comparator predicate *better*. (In the definition, $c\theta$ denotes the boolean result of applying the valuation θ to c, and we say that "$c\theta$ holds" if $c\theta = \textbf{true}$. Note that this is a specification of S, not an algorithm for computing it!)

$$\begin{aligned} S_0 &= \{\theta \mid \forall c \in H_0 \; c\theta \text{ holds}\} \\ S &= \{\theta \mid \theta \in S_0 \land \forall \sigma \in S_0 \; \neg \mathit{better}(\sigma, \theta, H)\} \end{aligned}$$

We now define the locally-predicate-better and weighted-sum-better comparators. As also noted above, we use an error function $e(c\theta)$ that returns a non-negative real number indicating how nearly constraint c is satisfied for a valuation θ. This function must have the property that $e(c\theta) = 0$ if and only if $c\theta$ holds. For any domain $\mathcal{D}$, we can use the trivial error function that returns 0 if the constraint is satisfied and 1 if it is not. A comparator that uses this error function is a *predicate* comparator. For a domain that is a metric space, we can use its metric in computing the error instead of the trivial error function. Such a comparator is a *metric* comparator.

The first of the comparators, *locally-better*, considers each constraint in H individually to find Pareto-optimal solutions.

Definition. A valuation θ is *locally-better* than another valuation σ if, for each of the constraints through some level $k - 1$, the error after applying θ is equal to that after applying σ, and at level k the error is strictly less for at least one constraint and less than or equal for all the rest.

$$\begin{aligned} &\mathit{locally\text{-}better}(\theta, \sigma, H) \equiv \\ &\quad \exists k > 0 \;\; \text{such that} \\ &\qquad \forall i \in 1 \ldots k-1 \; \forall p \in H_i \;\; e(p\theta) = e(p\sigma) \\ &\qquad \land \;\; \exists q \in H_k \;\; e(q\theta) < e(q\sigma) \\ &\qquad \land \;\; \forall r \in H_k \;\; e(r\theta) \leq e(r\sigma) \end{aligned}$$

Locally-predicate-better is then *locally-better* using the trivial error function that returns 0 if the constraint is satisfied and 1 if it is not.

Next, we define a schema *globally-better* for global comparators. The schema is parameterized by a function g that combines the errors of all the constraints H_i at a given level.

Definition. A valuation θ is *globally-better* than another valuation σ if, for each level through some level $k-1$, the combined errors of the constraints after applying θ is equal to that after applying σ, and at level k it is strictly less.

$$\begin{aligned}
&\textit{globally-better}(\theta, \sigma, H, g) \equiv \\
&\quad \exists k > 0 \ \text{ such that} \\
&\qquad \forall i \in 1 \ldots k-1 \ \ g(\theta, H_i) = g(\sigma, H_i) \\
&\qquad \wedge \ \ g(\theta, H_k) < g(\sigma, H_k)
\end{aligned}$$

Using *globally-better*, we now define *weighted-sum-better* by selecting a particular combining function g. The weight for constraint p is denoted by w_p. Each weight is a positive real number.

$$\begin{aligned}
\textit{weighted-sum-better}(\theta, \sigma, H) &\equiv \textit{globally-better}(\theta, \sigma, H, g) \\
\text{where } \ g(\tau, H_i) &\equiv \sum_{p \in H_i} w_p e(p\tau)
\end{aligned}$$

Different constraint solvers find solutions for different comparators. In our work to date, the two solvers we use that accommodate soft constraints are DeltaBlue, which finds locally-predicate-better solutions, and Cassowary, which finds weighted-sum-better solutions. (When there are infinitely many solutions, as in the above example with the two medium constraints $x = 0$ and $y = 0$, Cassowary has the additional property that it finds solutions that "tilt" toward one constraint or the other, so that it would find either the solution $x = 0$, $y = 10$ or else $x = 10$, $y = 0$ — but not for example $x = 3.6$ and $y = 6.4$, even though that is also a weighted-sum-better solution.)

In Babelsberg, the comparator to be used is thus named by the choice of constraint solver.

3.1 Read-only Annotations

A variable in a particular constraint (either hard or soft) can be annotated as *read-only.* Intuitively, when choosing the best solutions to a constraint hierarchy, constraints should not be allowed to affect the choice of values for their read-only variables, i.e., information can flow out of the read-only variables, but not into them. (Alternatively we can say that constraints are only allowed to affect the choice of values for their unannotated variables.) For example, consider the following constraints:

$$\begin{array}{rl}
\text{required} & x? + 5 = y \\
\text{medium} & y = 20 \\
\text{low} & x = 0
\end{array}$$

The solution is $\{x \mapsto 0, y \mapsto 5\}$ — the read-only annotation on the x in the required constraint prevents the solver from satisfying the $y = 20$ constraint instead of the $x = 0$ one.

A *one-way constraint*, as used in many solvers and systems, including the ubiquitous spreadsheet, can be represented by annotating all but one of the constrained variables as read-only, as in the following example:

$$\text{required} \quad x? + y? = z$$

Reference [BFBW92] presents a formal, declarative specification of read-only annotations, adapted from formalizations of read-only annotations in committed-choice logic languages [Mah87], but the above intuitive description is sufficient for the current purpose.

3.2 Support for State Change and Incrementality

When used in interactive graphical applications, as well as in other systems involving change over time, it is important to provide the constraint equivalent of "frame axioms" that specify that parts retain their old values as the system changes; without such constraints, we would often get counter-intuitive behavior, such as a constrained figure collapsing to a point. *Stay constraints* provide such a mechanism. As far as the theory of hard and soft constraints is concerned, a stay constraint is simply an equality constraint $v = c$ for variable v and constant c, typically with a low priority. Operationally, c will be the value of v at the previous time step.

To support incrementality, we also introduce the notion of an *edit constraint*. Formally, this again is simply an equality constraint $v = c$ for variable v and constant c. Operationally, it is used to model changing an input value, for example in response to the cursor position (where in this case v and c would both hold points: v would be a point in a diagram being moved, and c would be the cursor position). Edit constraints typically have a high (but not required) priority — the system will attempt to accommodate the edit action, but may be prevented from doing so, for example if a figure would leave the allowed display rectangle as a result of attempting to move it too far.

Both DeltaBlue and Cassowary treat stay and edit constraints specially, allowing very fast incremental re-satisfaction of a collection of constraints as new edit values stream into the system (and the weak stay constraints provide basic stability). Edit constraints prepare the internal data structures of the solver for fast resolving when specific variables change. For DeltaBlue, this involves pre-calculating the execution plan from the edit variables. For Cassowary, the Simplex tableau is set up so that it can be efficiently re-optimized given new values for the edit variables.

4 Object Constraint Programming

Babelsberg is an object constraint programming (ocp) language — the term *object constraint programming* is chosen to emphasize the integration with widespread object-oriented programming ideas, in particular methods, messages, and object encapsulation. This is in contrast to earlier work on cip, which emphasized separate, parallel concepts to specify and select behavior for constraints, rather than re-using the object-oriented behavior definitions. Similarly, systems such as Kaplan, BackTalk, and Ilog Solver expect the objects used in constraints to be special constraint-aware ones, rather than ordinary objects.

Our goals for Babelsberg include a syntax that is compatible with the base language. In our initial prototype, Babelsberg/R, the base language is Ruby, and the extensions are almost all semantic extensions, with only one minor (and completely optional) syntactic extension. This extension provides a more convenient syntax for writing read-only constraints. A subsequent implementation of Babelsberg in JavaScript — Babelsberg/JS — uses no syntactic extensions [FBH+14b].

The semantic model is also an extension of Ruby's, and supports all of the existing Ruby constructs such as classes, instances, methods, message sends, and blocks (closures). There are two semantic changes to support constraint creation and solving.

The first semantic addition is the inclusion of the *always* primitive — described in Section 4.1 — that receives a closure, and creates a constraint from it. The constraint is, that the last expression in the closure should return true. The closure can reference objects, invoke methods on them, express assertions about their identity, type, or protocol, and use assignments and control structures. However, some restrictions apply, and are described in Section 4.1.1.

The second semantic addition is the provision of *constrained variables* — described in Section 4.2 — whose values are defined by a constraint solver. Constrained variables can be changed by the solver to satisfy constraints when the system is perturbed, and assignments to them trigger resolving with the new value. Assignments may also fail, if that would make the constraint system unsatisfiable.

In this paper we provide an English-language description of the semantics of our constraints and the language changes. A formal operational semantics is in preparation, with a recent technical report [FMB14] presenting the step-by-step development of an operational semantics for a series of versions of Babelsberg, starting with a very simple language Babelsberg/Reals (in which the only datatypes are real numbers along with booleans for the constraint expressions) and through Babelsberg/Objects (a true object-oriented language, with mutable objects, classes, methods, messages, and inheritance, but without all the features and idiosyncrasies of Ruby or another practical, widely-used language).

4.1 Constraints as a Language Primitive

Our first semantic extension to the basic object oriented language is the addition of a new primitive, **always**. This primitive takes a closure argument: if the last expression in the closure evaluates to true the constraint is satisfied. (Usually, there would only be one expression.) The **always** primitive evaluates the closure using a modified interpreter, which collects all objects that contribute to the evaluation and creates a representation of the operations between them. This representation can then be passed to a solver to satisfy the constraint. (The syntax for **always** is the same as for a normal method call that accepts a closure argument.)

As a first example, consider the following class `TemperatureConverter` in Babelsberg/R, which maintains the appropriate relation between instance variables holding Centigrade and Fahrenheit values.

```
class TemperatureConverter
  attr_accessor :centigrade, :fahrenheit
  def initialize
    @centigrade = @fahrenheit = 0.0
    always { centigrade * 1.8 == fahrenheit - 32.0 }
    # constraint added and solver triggered during 'always'
  end
end
```

A constraint constructed through **always** is activated immediately. The primitive also returns a *Constraint* object that provides the interface to enable, disable, and inspect the constraint. We assign floating-point numbers to the two fields, so that Babelsberg/R uses a solver for floats based on their run-time type. However, we didn't need to give them values that satisfy the constraints — one or both are changed to keep the constraint satisfied.

If we make a new instance of `TemperatureConverter` and change either the Centigrade or Fahrenheit temperature, the other value will be changed as well to keep the constraint satisfied:

```
t = TemperatureConverter.new
t.centigrade = 100.0 # triggers solver, which changes t.fahrenheit to 212.0
```

In this example, the constraints are on the results of sending the messages `centigrade` and `fahrenheit`. These methods are just accessors for the corresponding fields. However, constraints can also include methods that perform other computations. For example, the rectangle from page 4 makes available its area as a computed property via the `area` method. Whether it is on screen is computed by the `visible?` predicate. In Babelsberg/R, we can use these existing Ruby methods in specifying constraints on a rectangle:

```
rect = Rectangle.new
always { rect.area >= 100 }
always { rect.visible? }
```

The first constraint says that the result returned from calling the `area` method should always be greater than or equal to 100, and if, for example, another part of the program assigns to the height of the rectangle, if necessary the width will be adjusted automatically to keep the constraint satisfied. The **`always`** primitive discovers that the `x` and `y` values of the `extent` object are multiplied to get the result of the `area`, and that this result should be greater or equal to 100. This relation is passed as a constraint to a float solver.

Similarly, if a negative location is assigned to the origin, it will be moved back to keep the rectangle visible.[1]

By placing the constraint on the result of sending messages rather than on fields, the system also respects object encapsulation. The values returned from the message sends in the rectangle example are both primitive types (float and boolean), but they can also be arbitrary objects. For example, we could add a constraint on the rectangle's center (a computed rather than a stored value, and a point rather than a primitive type):

```
always { rect.center == Point.new(100,100) }
```

4.1.1 Restrictions on Constraints

When the programmer uses a method in Babelsberg in a constraint, the underlying implementation generates a corresponding set of (generally simpler) constraints that can in turn be handed to an appropriate solver. Different language constructs in the methods can give rise to different sets of constraints, which may be more or less difficult for the solver. For example, disjunctions in the method give rise to disjunctions in the constraints sent to the solver, and of the currently provided solvers, only Z3 can accommodate these.

There are however three important restrictions on constraints that apply to all solvers.

First, evaluating the expression that defines the constraint should return a boolean — the constraint is that the expression evaluates to true.

[1]There are multiple possible locations that satisfy the `rect.visible?` constraint; here the system will move the origin as little as possible from the assigned location but so that the constraint is satisfied. The same holds for the area constraint. This behavior is a result of soft "stay" constraints. These are left implicit in this example, but can also be stated explicitly if desired. See Section 3.

Second, the constraint expression should either be free of side effects, or if there are side effects, they should be benign, e.g., caching. Also, repeatedly evaluating the constraint expression should return the same thing. (For example, an expression whose value depended on which of two processes happened to complete first wouldn't qualify.) This restriction is needed to provide the correct semantics for constraints.

Third, variables used in methods that will be called in a constraint must be used in a static single assignment (SSA) fashion. As an example of why this restriction is needed, consider the following method:

```
1 def bad_method(x)
2   sum = x
3   sum = sum+2
4   sum
5 end
```

Suppose we try to satisfy the constraint `10==bad_method(a)`. The system would then construct a constraint `sum==x` for the first line in the method, and `sum==sum+2` for the second. This is unsatisfiable, so an exception would be raised by the solver.

This last restriction can be removed if methods in which variables are multiply assigned are split into code blocks that are each in SSA form. Multiple assignments to the same variable are then rewritten as assignments to distinct variables (e.g., x_t, x_t+1, ...), which are connected via equality constraints. The above code, for example, can be split into two blocks (line 2 and lines 3-4), with the resulting constraints being `sum_1==x` and `sum_2==sum_1+2`. Currently, this rewriting must be done by hand by the programmer; in the future we plan to add an extension to do this code transformation automatically.

4.1.2 Constraints and Control Structures

Expressions to create constraints are simply statements in the host language, and so can appear in conditionals, loops, recursive methods, and so forth. Constraints themselves can also include iterations and conditionals. For example, here is a definition of the `sum` method for arrays that creates a set of addition constraints relating the array elements and their sum, using partial sums as intermediate variables.[2] Similar constructs can be used in representing more complicated structures.

```
class Array
  def sum
    inject (0) { |partial_sum, x| partial_sum + x }
  end
end
```

We ship a simple local propagation solver for constraints on float arrays. Constraints on the elements interact correctly with other constraints on the sum and values. For example, the programmer can use the solver to find a value for one element, given the sum and values for the others:

```
a = [0.0, 0.0, 0.0]
always { a[0] == 10 }
always { a[2] == 20 }
always { a.sum == 60 }
```

This gives the solution `a[1] = 30`. We also have constraints on the length of an array, and the array as a whole. For example, for arrays `a` and `b`:

[2]The `sum` method uses `inject` (known also as `reduce` or `fold`) instead of, for example, a call to `each` that repeatedly assigns to a local variable, so that we satisfy BABELSBERG's single assignment rule.

```
always { a.length == 10 }
always { a == b } # == is content equality for Ruby arrays
```

4.1.3 Constraints on Identity, Type, or Protocol

We can also write constraints on properties of objects such as their identity, class, and the messages that they respond to. For example:

```
always { x is? y  } # an identity constraint
# examples of constraints on the class or message protocol of an object:
always { a.class == Point }
always { a.kind_of?(Point) }
always { a.instance_of?(Point) }
always { a.respond_to?(:theta) }
```

Identity constraints, in our current implementation, are supported with a built-in local propagation solver. They are useful for defining data structures (for example a circular linked list) or when modeling aspects of the real world [LFBB94a]. For example, it is clearly different if two train tickets are valid for the *same* or an *equal* train.

For `a.class` and `a.instance_of?`, if `a` is already bound, the constraint is just a test; if it is unbound, then the constraint could be satisfied by creating a new instance of the appropriate class, binding it to `a`, and calling the `allocate` method of the appropriate class. Finally, `a.kind_of?` and `a.respond_to?` are only available as tests. (One could imagine satisfying `a.kind_of?` by backtracking through the possible classes of which `a` is an instance, but this seems complicated and without a clear use case. The situation for `a.respond_to?` is similar.)

Note that a benefit of this design is that type declarations or their equivalent are just constraints.

4.1.4 Soft Constraints

As noted in Section 3, soft constraints are useful, for example, in interactive graphical applications. Although not generally required for CIP languages, most support soft constraints. As an example, a soft constraint $a + b = c$, with priority `high`, would be written in Babelsberg/R as

```
always(priority: :high) {a+b==c}
```

In Babelsberg, whether soft constraints can be accommodated and how soft constraints are traded off are considered properties of the solver. Solvers that cannot accommodate constraint priorities should throw an exception when passed a priority. Some solvers provide means of simulating soft constraints, for example, Z3 can produce "unsatisfiable cores", which is a list of constraints that cannot be simultaneously true. Soft constraints in the list can be disabled to try again without them. However, this provides two priorities (required and not), and is slower, because the solver has to solve at least twice[3].

In our current implementation, we provide a mapping of the symbolic priorities `low`, `medium`, `high`, and `required`, for the Cassowary and DeltaBlue solvers. (Our initial experience with Babelsberg/R, and considerable prior experience with Cassowary and DeltaBlue, imply that only a small number of priorities are needed, and tend to be used in stylized ways [BBS01].)

[3] An upcoming version of Z3 will support soft constraints with arbitrary priorities and optimization directly.

4.2 Constrained Variables

Our other semantic extension concerns the way assignment is handled in the object constraint programming language. In an OCP program, variables that are used in constraints become *constrained variables*. Each variable is determined by at most one solver, and the variable value is stored in a solver specific data structure. Reading such variables returns solver determined values and assigning to such variables triggers the solver and may raise an exception, if the assignment would violate a required constraint.

The **`always`** primitive executes its closure argument using a modified interpreter that tracks which objects contribute to the constraint. Each variable that is accessed in the execution of the closure is converted into a constrained variable. Then, depending on the run-time type of the variable, the interpreter checks which of the available solvers can solve constraints over this type, e.g., Cassowary for floats. If a suitable solver is available, the variable is replaced with a solver-specific object that represents this variable in the solvers' data structures. This solver object replaces the actual value of the variable.

A solver object should respond to the subset of the interface of the associated OO value that its solver can solve. For example, Cassowary can solve linear constraints over float values, so the Cassowary variables returned for floats respond to the methods `+`, `-`, `*`, and `/`, but not `sin` or `**`. If a programmer tries to assert a constraint using those latter methods, an exception will be raised.

Constrained variables for which no solver exists are used as constants in the execution of the constraint expression. When these variables change, the constraint expression has to be recalculated entirely.

Assignments to variables in constraints assert equality constraints. This supports constructing new objects in the predicate that connect values, but this also means that all code blocks encountered during constraint construction must be in single assignment form (Section 4.1.1).

4.2.1 Reading and Writing Constrained Variables

When constrained variables that were replaced with solver objects are read, their actual value is retrieved from the solver — it is up to the solver to convert the internal representation into a representation suitable for the OO language (e.g., converting reals into floats). Variables that were used in constraints, but not replaced with solver objects, are read normally.

Assigning to a constrained variable also distinguishes these two cases. If the variable was replaced with a solver object, instead of storing the supplied value directly, a required equality constraint between the variable and the new value is temporarily added to the solver and the solver is invoked. If the solver can satisfy the constraints (including the new equality), the assignment succeeded; if not, an exception is raised. Afterward, the equality constraint is removed.

In the second case, the variable is used in a constraint, but no solver is available to determine its value (e.g., a string that was used in a constraint when no string solver was available). The value of such variables is treated as constant in the constraint. If the variable is assigned, all constraints it participated in are disabled, and their closures are re-executed to create new constraints. This, too, may lead to a situation where some solvers cannot satisfy their constraints. In that case, the old constraints are re-enabled, the assignment is undone, and an exception is raised. Undoing the assignment is necessary to ensure that no inconsistent system state arises – an assignment either works or fails completely.

4.2.2 Stay Constraints

In imperative languages, programmers expect their variables to retain their values, unless some assignment operation changes them. Section 3.2 describes how constraints interact with state change. For example, in interactive graphical applications, when moving one part of a constrained figure, the user generally expects other parts to remain where they are unless there is some reason for them to change to satisfy the constraints. A desire that something remain the same if possible is represented as a *stay constraint*, which may have an associated priority. For example, this stay constraint says that we prefer that x keep its value, if possible, when satisfying other constraints:

```
always { x.stay(:low) }
```

For those solvers that support soft constraints, Babelsberg automatically adds a lowest-priority stay constraint to every constrained variable so that it keeps its old value if possible when satisfying the other constraints. This also directs the solver to find a solution that is close to the prior value, if a variable does have to change.

4.2.3 Read-only Variables

Some variables should not be modified by the solver (for example, in an interactive application, dragged objects should follow the mouse, not vice-versa). To express this, Babelsberg includes support for read-only variables as described in Section 3. In Babelsberg/R, variables are marked as read-only by sending the question mark method (*?*) to them in a constraint expression. Ruby does not normally allow the question mark as a method name — instead, Babelsberg/R extends Ruby's syntax to support this. (Having a special syntax for this is purely a convenience rather than being an essential aspect of Babelsberg/R, and the JavaScript implementation of Babelsberg [FBH+14b] uses a global function `ro` instead.)

Read-only variables are useful, for example, for parameterized constraints so the solver knows not to change the parameter to satisfy the constraint:

```
class Rectangle
  def fix_size (desiredsize )
    always { self.area == desiredsize.? }
  end
end
```

Without the annotation, a solver might change the local variable `desiredsize` rather than the receiver.

4.2.4 Incremental Resatisfaction

Some applications involve repeatedly re-satisfying the same set of constraints with differing input values. A common case is an interactive graphical application with a constrained figure, where we move some part of the figure with the mouse. For such applications, it is important to re-solve the constraints efficiently. Section 3.2 introduced *edit constraints*, which prepare the internal state of a solver for rapid re-satisfaction. In Babelsberg/R, edit constraints must be used explicitly with a particular solver. An `edit` method is provided as part of the Cassowary library, another is included with DeltaBlue. These methods use the meta-level application programming interface (API) of the `Constraint` class to query which variables participate in a constraint, create edit constraints for them, and feed new values into the resulting edit variables. Other libraries with support for incremental resolving can provide similar methods. Users thus can take advantage of incremental solvers when available to re-satisfy constraints very quickly.

The provided `edit` method takes a variable to be edited, a stream (an object that responds to **`next`**) that provides the new values, and optionally a priority. (The priority defaults to the highest non-required one.) The solver then repeatedly re-solves the constraints given the new values for edit variables. If the edit constraint is required, the system raises an exception if the constraints cannot be satisfied with the new edit value. Finally, when the stream is empty, it removes the edit constraint and returns. Note that this works if the stream is being fed from another process that provides new values only when available — the process that has the edit constraint should just block until one is available.

For example, suppose we make an instance of `TemperatureConverter`. Then we can give 100 new values to `centigrade`, and have `fahrenheit` change correspondingly each time (this is useful, for example, if we combine this with a graphical slider that a user can drag to input new values):

```
converter = TemperatureConverter.new
enumerator = (0..99).each
edit(enumerator) {converter.centigrade}
```

The stream need not contain only primitive types — a common case in interactive graphics is a stream of new point values for a location of something being moved. The implementation of edit constraints is discussed in Section 5.3.

4.3 Solving Constraints

Given a set of constraints, we need to find a solution to them. Babelsberg provides an architecture that supports multiple constraint solvers, that makes it straightforward to add new solvers, and that doesn't privilege the solvers provided with the basic implementation. (They are simply the solvers that are in the initial library.) In the current implementation of Babelsberg/R, the available solvers are Cassowary, Z3, a solver for float and integer arrays, and DeltaBlue.

Currently, programmers must explicitly indicate which solvers are available in a given program, just as they choose other libraries appropriate for the application. Solvers register themselves for specific types, and are then chosen eagerly based on the run-time type of the variables that occur in a constraint. Only one solver can be registered for a particular type.

In cases where different solvers can handle the same types (for example, both Z3 and Cassowary handle constraints on floats in Babelsberg/R), the programmer has to decide which solver to use by activating only the desired one. The simplest way to do this is to only include the desired library. There is also the option to provide the solver as an additional argument to **`always`**. In some cases, this choice may simply be a matter of preference, however, other restrictions apply. For example, since Cassowary cannot solve nonlinear equations, the developer should select Z3 instead:

```
always(solver: Z3::Instance) { a ** 2.0 < 16 }
```

Developers must be aware that solvers may represent types differently internally. For example, Z3 works on reals, but its C API represent results as double-precision floating point numbers, so rounding errors may occur.

For opaque data structures (such as an SDL[4] window), the solver cannot set properties directly, but has to use the API instead. In that case, a local propagation solver like DeltaBlue must be used and, besides a predicate, a list of setter expressions must be provided. The following constraint will make a rectangle fill the SDL window, or the window will be shrunk to fit the rectangle.

```
always(solver: DeltaBlue::Solver:: Instance,
        priority :  :high,
        predicate:  {window.extent == rectangle.extent}) {[
  window.extent <-> { window.set_video_mode(rectangle.extent) },
  rectangle. extent <-> { rectangle.extent = window.get_video_mode }
]}
```

Additional arguments to `always` are passed to and interpreted by the solver. In this case, the `predicate` and `priority` arguments are passed to DeltaBlue and not interpreted by Babelsberg. In the constraint closure, the list of setters is provided by connecting the target variable and the setter expression using the `<->` syntax.

Developers might write constraints that are too difficult to solve, or for which no solver exists. In that case, the constraint expression is simply evaluated as a test that is checked by the runtime whenever a variable changes that may change the result of the test. If the test fails, an exception is raised explaining that no solver for the constraint was available.

We are currently working on adding support for cooperating constraint solvers, so that several solvers can work together to find a solution to constraints that connect variables from different domains [Bor12].

4.4 Constraint Duration and Activation

Constraints have *durations* during which they are active. The `always` duration declares that its constraint becomes active when the `always` statement is evaluated and remains active indefinitely after that, unless explicitly deactivated.

As mentioned in Section 4.1, the `always` primitive returns a *Constraint* object that provides meta-level access to control constraint activation, allowing it to be subsequently deactivated and reactivated. Babelsberg provides two additional methods that provide more structured control of constraint durations. The `once` method activates a constraint, satisfies, and then retracts it. This is useful to initialize a system state to satisfy a constraint. The `assert-during` construct takes a closure and activates a constraint for the duration of the evaluation of the closure.

A related issue is *when* constraint satisfaction is invoked. Babelsberg's default behavior is that constraints are immediately satisfied or re-satisfied whenever there is a change to one of the constrained variables. Sometimes this is not the desired behavior, for example, when there is a sequence of assignments that change the state of an object, with the object being in a temporarily inconsistent state in the midst of the assignments. To handle this, Babelsberg includes a construct to allow a sequence of assignments to be made, with solving invoked only after all the assignments have been made. Ruby provides multi-assignments to store values into multiple variables in a single statement, and this is used in Babelsberg/R to implement this construct. If multiple variables that have constraints on them are assigned using multi-assignment,

[4]`http://libsdl.org/`

all values are assigned before the solver is triggered. If any constraints cannot be satisfied, all assignments are immediately undone, again, to ensure that assignments either succeed or fail entirely.

5 Implementation

The Ruby virtual machine (VM) we use as a basis for BABELSBERG/R is *Topaz* [GF14], an experimental VM built using the *PyPy/RPython* toolchain [RP06]. While Topaz is a Ruby interpreter, the RPython toolchain provides it with a fast JIT compiler and garbage collector. We have extended the interpreter, and inherit the JIT and garbage collector from RPython. In this section, we first provide an overview of the key features of the implementation before plunging into the details.

The Topaz Ruby VM is written in a restricted, object-oriented subset of Python called RPython. Topaz executes Ruby by compiling Ruby source code into bytecodes and interpreting the bytecodes. The interpreter is written in an object oriented fashion, with each bytecode implemented as a method of the class `Interpreter`. Topaz also represents Ruby language constructs such as references and scopes as instances of RPython classes. As an example, local variables are represented by `Cell` objects, which provide methods to access and update the variable's value. In BABELSBERG/R, we leverage the object-oriented design of Topaz to implement OCP.

The changes we made to the Topaz VM are two-fold. First, we modified the interpreter to support a *constraint construction mode* (cf. Section 5.2) and a primitive to enter this mode. Second, we extended the cells to support *constrained variables* (cf. Section 5.1) by allowing the same name to refer to multiple objects, one an object-oriented value and the other a *solver object*. (Note that this use of the same name to refer to multiple objects is a feature of the implementation only — it is not visible to the programmer.)

5.1 Constrained Variables

Ruby provides five types of variables: locals, instance variables, class variables, globals, and constants. (While constants should be assigned only once, this is simply a convention in Ruby, and is not checked by the interpreter.) Of these variable types, we allow three as constrained variables: locals, instance variables, and class variables.

Ordinary variables are converted automatically to constrained variables if they are used in a constraint expression. A constraint expression is passed to the **`always`** primitive as an ordinary block closure. All locals referenced in the expression are stored as cell objects, which in BABELSBERG/R have an additional field to store solver objects. Instance and class variables in Topaz are stored in maps [CUL89]. For BABELSBERG/R, we have extended these to store solver objects as well as ordinary objects, so that once the JIT has warmed up, read access to solver objects is no slower than access to ordinary objects.

Converting ordinary variables into constrained variables has an impact on garbage collection. Solver objects usually have more references pointing to them than ordinary objects — besides their scope and owner (in the case of instance or class variables), solver objects are also referenced from one or more constraints. Because solver objects can also be implemented in the host language, this means that they may only be garbage collected if no more active constraints refer to them.

5.2 Execution Contexts

To implement the different execution modes — *imperative execution*, and *constraint construction* — we extended the default interpreter, and added `ConstraintInterpreter` as a subclass. The `ConstraintInterpreter` changes how locals, instance variables, and class variables are accessed.

5.2.1 Imperative Execution

This is the normal execution mode, as implemented in the Topaz interpreter. We have extended the STORE and LOAD bytecode methods with an additional check whether a variable refers just to an ordinary object or to a solver object as well. In the latter case, instead of reading or changing the value of the variable, the solver object for the variable receives the messages `value` or `suggest_value`, respectively. As long as all constraints are satisfied, this difference is not visible to the programmer. However, while direct, destructive assignment to a normal variable in imperative programs always succeeds, `suggest_value` triggers one or more solvers. If any solver fails to satisfy its constraints using the new value, an exception is raised. This exception is propagated by the VM, and should either be handled by the programmer or allowed to halt execution of the program. If an assignment fails in this way, the variable retains its original value.

As described in Section 4.4, in BABELSBERG/R, if multiple variables that have constraints on them are assigned using multi-assignment, all values are assigned before a solver is triggered, and all assignments are undone if an exception is raised during solving. This way, if the exception is caught, the program is not in violation of any constraints.

5.2.2 Constraint Construction

The initializer for constraint objects — usually called through **`always`** — activates an execution context for constraint construction. In this mode, variable values are replaced with their associated solver objects in LOAD instructions. These solver objects are created by sending `for_constraint` to the current variable value. This method should return an object that responds to `value` and `suggest_value`. The `value` method extracts the value from a solver's internal representation, while `suggest_value` sends the candidate value to a solver and requests that it re-satisfy the constraints. For values that do not respond to `for_constraint`, a generic solver object is returned that removes and recalculates all constraints in which it occurs whenever its value changes.

STORE instructions in this mode create equality constraints. This is necessary to support constructing new objects in the predicates that connect values (for example, a 2d point with x and y values that were constructed in the constraint.) However, this also means that all code blocks encountered during constraint construction must be in single assignment form (cf. Section 4.1.1).

To support solvers written in the host language, the VM needs a way to distinguish code that should be executed in this mode from code that should not. To support this, solvers written in the host language should be subclasses of `ConstraintObject`. This class serves as a marker to leave constraint execution mode when we enter solver code.

Constraint Solving This mode is simply a flag that is active whenever code runs in the dynamic extent of a method of a `ConstraintObject`. It prevents nested sends of `suggest_value` from causing recursive calls to the solver. This is necessary to support

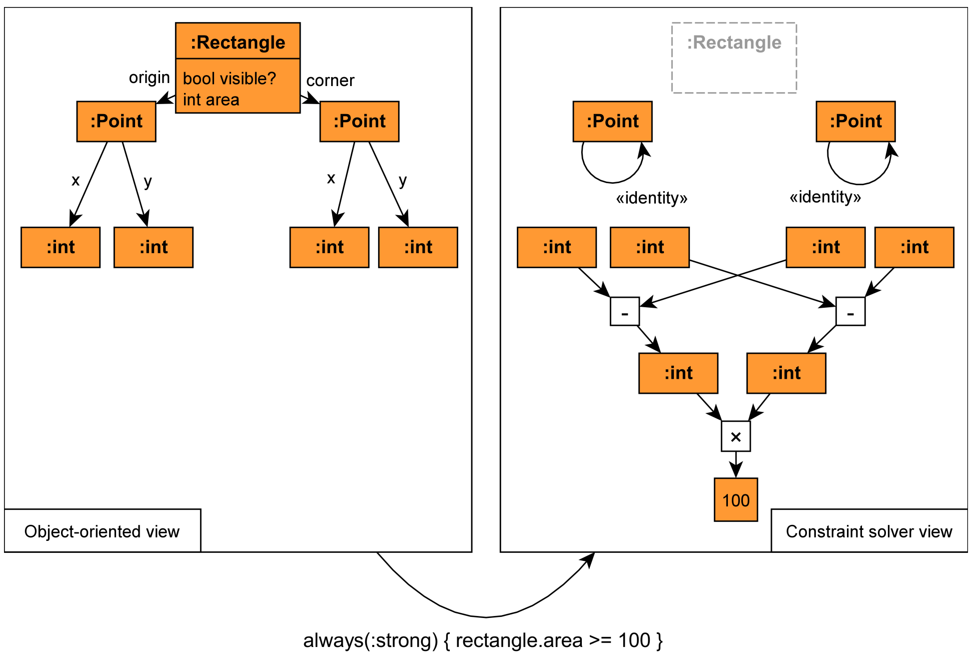


Figure 1 – Objects are connected through instance variables. When a constraint is constructed, their parts become connected through constraints.

solvers written in Ruby, but implies that solvers themselves cannot use constraints in their implementation.[5]

Constraint Construction Example

To illustrate how our implementation supports the combination of objects and constraints, consider the rectangle example in Babelsberg from Section 4. The code asserts that the area of the rectangle `rect` should always be greater than or equal to 100. The assertion is expressed by sending the `area` method to `rect`, and then sending the `>=` method to the result. The only variable named explicitly in the constraint is `rect`, but there are other variables that play a role in it.

In constraint construction mode, the `rect` variable is replaced with a generic solver object, since the class `Rectangle` does not implement the method `for_constraint`. This placeholder delegates the message `area` to the rectangle. During the execution of the `area` method, the points are replaced with generic solver objects, and their x and y values (floats) are replaced with specific solver objects (for example Z3 variables, cf. Figure 1). In this mode, the messages to the float values return symbolic expressions rather than calculating the current area of the rectangle. The expression representing the area of the rectangle is then sent the message `>=` with 100 as its argument and returns an inequality constraint.

The constraint construction is complete when the block passed to `always` returns. The values and relations among them produced by this symbolic execution are gathered into a *Constraint* object containing specific solver objects and generic solver objects.

[5]If we wanted to support other cooperating paradigms in addition to constraint-oriented programming, we would generalize this technique to support additional execution contexts.

Specific solver objects are objects that are connected in a way that the solver can reason about (e.g., floats connected via equalities or inequalities or arithmetic relations, or booleans connected via boolean operations). Generic solver objects are all other objects that contribute to the constraint but about which the solver cannot reason directly.

The constraint solvers operate on specific solver objects directly to solve the constraint. Changes made by the solver to these objects show up in the OO view on the next access to the variables from imperative code, at which point their values are copied to their OO counterparts. Assignments to any variable encountered during the constraint construction will trigger the solver to re-satisfy the constraint (and potentially raise an error if the assignment is inconsistent with the constraint).

Generic solver objects invalidate the constraint if their OO value changes through imperative assignment. This invalidation retracts all solver objects created during constraint construction and re-executes the block to create new solver objects and constraints. This means that the block may be re-executed multiple times during the run-time of the program. (This is one reason why any side-effects in constraints should be benign, as noted in Section 4.1.1.)

5.3 Implementing Edit Constraints

Edit constraints are used to support incremental constraint satisfaction, and are important for achieving good performance in interactive applications. The `edit` method provided as part of BABELSBERG/R's Cassowary library adds edit constraints and repeatedly updates them with values from a stream.

Cassowary as shipped in the BABELSBERG/R standard library allows the variables and stream to hold user-defined objects as well as primitive types. The library uses the API of the `Constraint` class to access the constraint values associated with variable names, creates edit constraints for them, and feeds the stream into the resulting edit variables.

To use Cassowary as the solver, we need edit variables that are Floats (e.g., the x and y values of a midpoint), but we also want to do this in an object-oriented way that respects encapsulation. To support this, the client passes an array of method names for the return values that should be updated in the edit constraint (e.g., `x` and `y` for a point — those values may be calculated or direct accessors.) Cassowary creates new edit variables, and adds an equality constraint to the return values of the methods. Thus, the internal storage layout of the class is not visible to the programmer from outside the object, because the equality constraint is simply asserted on the results of message sends using the **`always`** primitive.

In the following example, the mouse locations or the mouse point might store their x and y values directly, or might be points represented using polar coordinates. In either case, the edit constraints apply to the return values of their respective `x` and `y` methods:

```
edit(stream: mouse.locations.each, accessors: [:x, :y]) { mouse_point }
```

In a DeltaBlue-specific edit method, the edit constraints returned could be simpler, since DeltaBlue local propagation methods can apply to user-defined objects such as points, not just to floats. The point would be simply updated rather than dealing with its x and y coordinates separately, and the data flow plan would update the objects constrained to be equal to the point that represents the mouse location.

5.4 Adding New Solvers

During constraint construction, the VM sends the `for_constraint` message to each variable value encountered during the execution. User code can add solvers to the system by dynamically adding a `for_constraint` method to those classes for which the solver is applicable, making use of Ruby's open classes. This method takes the name under which the variable is accessed as an argument, and should return an object that implements a subset of the interfaces that the solver can reason about, as well as the methods `value` and `suggest_value` (as described in cf. Section 5.2.)

The Cassowary solver extends the Float class:

```
  def for_constraint(name)
    v = Cassowary::Variable.new(name: name, value: self)
    Cassowary::SimplexSolver.instance.add_stay(v)
    v
  end
```

This method creates a new variable, adds a low-priority stay so the solver attempts to keep the value stable, and returns the constraint object. The VM then sends messages to this object instead of the Float object in the context of the constraint execution.

In the case of immutable objects (such as floats and integers), the VM directly returns the variable value determined by the solver. However, for mutable objects, solver libraries can provide the method `assign_constraint_value` to update the ordinary object. Babelsberg/R provides a solver over numeric arrays, which uses `assign_constraint_value` to update the array contents. In the `for_constraint` method for arrays, it returns a `NumericArrayConstraintVariable` if all elements in the array are instances of a subclass of `Numeric`.

```
class Array
  def for_constraint(name)
    if self.all? { |e| e.is_a? Numeric }
      return NumericArrayConstraintVariable.new(self)
    end
  end

  def assign_constraint_value(val)
    self.replace(val)
  end
end
```

This illustrates how the solvers provided by Babelsberg/R are simply constraint solver libraries that extend core classes. The `NumericArrayConstraintVariable`, for example, allows element access, provides the Ruby collection API (each, map, inject, ...), the `sum` method (which calculates the sum of elements) and the message `length`, to solve constraints over the length of arrays.

Below is an example that asserts constraints on TWP-encoded[6] short strings (represented as an array of bytes). TWP short strings are at most 110 bytes, with the first byte giving the length of the string plus a tag. Line 3 constrains this particular string to contain only capital letters (the `?A` Ruby syntax gives the byte value of a character).

```
1 twp_str = []
2 always { twp_str.length == twp_str[0] + 17 }
3 always { twp_str.length <= 109 }
4 always { twp_str.each {|byte| byte >= ?A && byte <= ?Z} }
```

[6]`http://www.dcl.hpi.uni-potsdam.de/teaching/mds/twp3.txt`

6 Evaluation

We evaluate BABELSBERG and BABELSBERG/R first with respect to performance, and second with respect to how well the language supports desirable semantic properties for an object constraint language.

6.1 Performance Evaluation

There are four questions we are interested in regarding performance: a) how does constraint solving performance compare with imperative code for satisfying constraints, b) how does BABELSBERG/R performance compare with other constraint-based languages, c) how is performance of pure object-oriented code affected by our extensions to the VM, and d) how does a practical application perform if refactored to use constraints.

Constraint Solving To answer the first question, we used an example from Kaleidoscope'93 [LFBB94c] and adapted it to BABELSBERG/R (cf. Figure 2). In this example, the user drags the upper end of the mercury in a thermometer using the mouse. However, the mercury cannot go outside the bounds of the thermometer, even if the user tries to drag it out. Additionally, a gray and white rectangle on the screen should be updated to reflect the new mercury position, and a displayed number should reflect the integer value of the mercury top. Refactoring the imperative version for BABELSBERG makes it more general, so the comparison is biased towards the imperative code. However, this example demonstrates the performance impact if an imperative program is refactored with the goal to make it more readable, not more general.

Note that the object-constraint version may be written in two ways: one that is more like the imperative version and assigns new mouse locations in a loop; and a more constraint-oriented version that declares `mouse.location_y` as an edit variable that triggers incrementally re-satisfying the constraints. The latter is expected to be much faster, as Cassowary can just re-optimize a previously optimal solution.

Table 1 shows that the naive object-constraint version is usually many hundreds of times slower than the purely imperative solution. That is because the imperative version is executing mostly machine code after a few thousand iterations, but the

```
Iterations . times do |i|
  mouse.location_y = i
  old = mercury.top
  mercury.top = mouse.location_y
  if mercury.top > thermometer.top
    mercury.top = thermometer.top
  end
  temperature = mercury.height
  if old < mercury.top
    # moves upwards (draws over the white)
    gray.top = mercury.top
  else
    # moves downwards (draws over the gray)
    white.bottom = mercury.top
  end
  display. number = temperature
end
```

(a) Imperative version

```
always { temperature == mercury.height }
always { white.top == thermometer.top }
always { white.bottom == mercury.top }
always { gray.top == mercury.top }
always { gray.bottom == mercury.bottom }
always { display.number == temperature }
always(:high) { mercury.top == mouse.location_y }
always { mercury.top <= thermometer.top }
always { mercury.bottom == thermometer.bottom }
always { thermometer.bottom == 0 }
always { thermometer.top == 200 }

Iterations . times do |i|
  mouse.location_y = i
end

# edit(Iterations. times) { mouse.location_y }
```

(b) Object-constraint version

Figure 2 – Interactive thermometer example from [LFBB94c]

Iterations	Imperative	Constraints	Edit Constraints
100	1(σ0.241)	36.1(σ3.14)	6.24(σ0.828)
10,000	1(σ0.14)	629(σ6.91)	7.72(σ0.526)
1,000,000	1(σ0.12)	45137(σ458)	52.5(σ2.24)

Table 1 – Thermometer benchmark (normalized against imperative version)

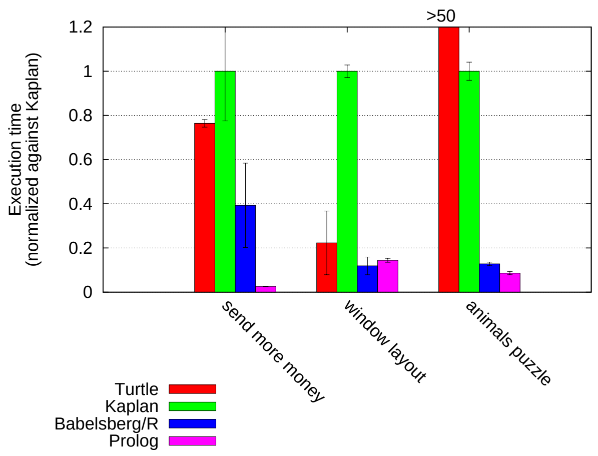


Figure 3 – CIP benchmarks

constraint version has many more loops which the JIT is struggling to optimize. Using edit constraints, repeated solving of constraints is much faster. Keep in mind that these results use a solver that is also written in pure Ruby, which is by itself orders of magnitude slower than a C++ based version.

Performance Across Different Languages Unfortunately, good benchmarks to measure the intended usage of languages that integrate constraints with imperative programming are difficult to find. Papers for such languages that evaluate performance use classical constraint solving problems. For our comparison, we selected three such problems: the *Send-More-Money* cryptarithmetic problem [Dud24], a layout example from the Turtle distribution (without preferential constraints), and the *Animals*[7] puzzle. We ran these benchmarks on Turtle, Kaplan, and BABELSBERG/R as examples of CIP languages; as well as on SWI-Prolog's CLP library, to compare the performance of constraint-imperative with constraint-logic programming.

Each benchmark was run in a loop to allow the JIT in Kaplan and BABELSBERG/R to warm up, and the executions were repeated 10 times.

Figure 3 shows that the CIP languages are comparable in performance, but Prolog with a constraint satisfaction library is a better choice for these problems when no imperative features are used. Both BABELSBERG/R and Kaplan use Z3 in these benchmarks. The difference in performance between them may be due to the overhead of creating the constraints, but we have also found Z3 performance to sometimes vary wildly between releases. Because Kaplan ships an older, modified version of the Z3

[7]Spent \$100 to buy 100 pets. You must buy at least 1 of each pet. Dogs cost \$25, cats \$1, and mice \$0.25.

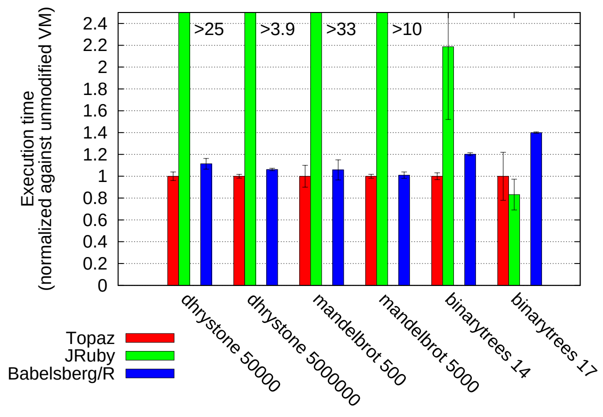


Figure 4 – Metatracing VM benchmarks

library than Babelsberg/R requires, we have not been able to compare them using the same version of Z3.

Performance of Pure Object-oriented Code For imperative code performance, we ran a number of tests from the metatracing VMs experiment [BT13] against the unmodified *Topaz* Ruby VM and the *JRuby* VM.

For purely imperative code, Babelsberg/R is consistently less than 1.5 times slower in these benchmarks than the unmodified VM (Figure 4). The only benchmark for which Babelsberg/R is significantly worse than Topaz is *Binarytrees*. Binarytrees is a strongly recursive benchmark, at which Topaz' (and thus Babelsberg/R's JIT) is bad at optimizing (which is why JRuby does better for this benchmark). The overhead in Babelsberg/R is due to an additional test each time a variable is accessed, to check whether this variable participates in a constraint. An annotation in the VM [BCF+11] allows the JIT to remove this test when there are no constraints on variables in the loop. The JIT consequently removes the check if it is not needed, so all the overhead we are seeing in these results is in code that has not been compiled by the JIT. Binarytrees is especially bad here, because very little code is compiled by the JIT for this benchmark.

Considering that JRuby and Topaz are, for these benchmarks, currently the fastest Ruby VMs, we consider Babelsberg/R to be very performant for purely imperative code.

Application Example We have written a simple video streaming application. This application reads a folder with bitmaps and streams raw video to a video player. The quality with which the video is streamed depends on a number of variables:

User Preference Users may decide not to overload their systems and explicitly choose a lower quality. This should be an upper bound for the quality of the streaming.

Encoding Time The program has only 1/25 second to encode a frame. If it takes longer, the quality should be automatically reduced to ensure smooth playback.

System Load The overall system load of the encoding system should be less than 80%, to retain sufficient resources for other tasks.

Quality Bounds Whatever users choose as preferred quality, the quality cannot be below 0 or above 100%.

In the imperative version of the application, these constraints were explicitly checked and re-satisfied after each frame. The object-constraint solution uses custom solvers for File contents and method execution time. Both versions could stream video at the highest quality setting to up to 8 clients on an Intel i5 2.4 GHz CPU. Both reduced the quality afterwards. Because most of the time was spent encoding and streaming video, both versions performed equivalently well. Most code was shared between the versions, only the code to adjust the quality was replaced with constraints.

Both versions spent on average 43.9 ms encoding each frame and sending it to the client. Also for both versions, 0.064 ms were spent fetching the current load and reading the preference file. The imperative version spent an additional 0.461 ms adjusting the quality after each frame, whereas the object-constraint version used 2.23 ms for that. In relation to the frame encoding time, the impact of constraint solving was small, and the imperative code managed to encode and play 22.5 frames per second on average versus 21.6 fps for the object-constraint code.

VM Hooks for Customizing Constraint Construction This benchmark also demonstrates how the `for_constraint` and `assign_constraint_value` VM hook methods are useful not only for solvers, but also when programmers want to provide a particular interpretation of certain application domain objects in their constraints. We used this express constraints on the contents of a preference file and the last execution time of the frame encoding method in our constraints.

The effort to allow Cassowary to work with file content and method execution time was small. The following class definition was enough for us to be able to use the execution time of a method in a constraint:

```ruby
class MethodTimer
  def initialize (klass, symbol)
    time = 0
    @constrained_time = Constraint.new { time }

    old_method = klass.instance_method(symbol)
    klass.define_method(symbol) do |*args, &block|
      start = Time.now
      res = old_method.bind(self).call(*args, &block)
      time = Time.now - start
      res
    end
  end

  def for_constraint(name)
    @constrained_time.?
  end
end
```

A `MethodTimer` is instantiated with a class and the name of a method. It wraps this method to record its execution time. When used in a constraint, a read-only constraint object is returned that is connected with the `time` local variable. Because it is read-only, the `assign_constraint_value` method is omitted. For the configuration file object, the `assign_constraint_value` method is as follows:

```
def assign_constraint_value(float)
  if float != @content.to_f
    raise "cannot assign to read−only file" unless @writable
    @file.truncate(0)
    @file.rewind
    @file.write(float)
  end
end
```

This illustrates that our VM hooks for constraint construction provide a limited local propagation mechanism to interpret complex objects in the type domain of a particular solver.

6.2 Comparison with other approaches

Babelsberg has a number of properties that we regard as desirable for an object-constraint programming language. Table 2 shows these properties and compares OCP to related approaches presented in Section 2. As a continuation of CIP, OCP shares most properties with languages like Kaleidoscope and Turtle.

Unified Language Constructs Programs in Babelsberg appear as ordinary OO programs if no constraints are used, but can be easily adapted to use constraints where it makes sense. Programmers re-use the object-oriented method definitions to specify constraints, thus respecting object encapsulation. Furthermore, techniques such as inheritance and dynamic typing operate correctly with constraints.

In contrast, library and DSL based approaches separate constraints from imperative code through a different syntax and semantics. For example, in languages such as Kaplan or Ilog Solver, developers must explicitly wrap objects in classes meant for use in constraints. Functional reactive programming (FRP) and CIP languages use *propagation hooks* and *constraint constructors* respectively to support constraints (propagation hooks are the functions that compute new values for connected variables, while constraint constructors are Kaleidoscope's mechanism for converting constraints on user-defined objects into primitive constraints for a solver.)

Automatic Solving Using libraries for constraint satisfaction allows programmers to write code that (intentionally or unintentionally) circumvents previously asserted constraints. Approaches that integrate constraints at a language level do not allow such circumvention, and attempt to re-satisfy constraints whenever they are violated during program execution.

Linguistic Symbiosis D'Hondt et al. [DGJ04] argue that linguistic symbiosis between different programming paradigms is required to support the evolution of programs

	Libraries	DSL	Dataflow/FRP	CIP	OCP
Unified Language Constructs					x
Automatic Solving		x	x	x	x
Linguistic Symbiosis			x	x	x
Extensible Solvers	x		(x)	(x)	x
Suitably Expressive Constraints	x	x		x	x
Performant Pure-OO code	x	x	x		x

Table 2 – Comparison of OCP with related work

from the object-oriented paradigm to a constraint-oriented solution and vice versa. DSL and library based approaches do not support such incremental refactoring between paradigms as well as approaches in which constraints are written in the host language.

Extensible Solvers Libraries provide the most flexibility for choosing different solvers depending on programmer needs. FRP languages can, to some extent, be combined with solver libraries to achieve a comparable flexibility. CIP languages also provide a more controlled way for developers to use different solvers by writing constraint constructors that reformulate constraints using a different solver.

In BABELSBERG, all solvers use the same interface to communicate with the VM so developers can add new solvers and replace existing ones to support new type domains, or to use solvers that give better results or performance for a particular problem.

Suitably Expressive Constraints To take advantage of the constraint paradigm, the language should allow a rich set of constraints to be written and solved. Multi-directional constraints are important for some applications, while others additionally require solvers that can accommodate simultaneous equations and inequalities. (For example, to simulate a Wheatstone bridge requires solving simultaneous equations, while the video streaming example needs inequalities as well as equalities, and both hard and soft constraints.) On the other hand, given an overly powerful but slow solver, it becomes all too easy to write constraints that take a very long time indeed to solve or that are intractable. We believe that BABELSBERG strikes an appropriate balance here, by providing an expressive set of constraints with the solvers in the initial library, and by allowing more powerful solvers to be added if desired. However, much more experience is needed to test whether in fact this is an appropriate balance, and to adjust it as needed; and as noted in Section 7, an important direction will be adding better support for debugging, explanation, and benchmarking.

Performant Pure OO Code Kaleidoscope provided a declarative semantics for assignment, type declaration, and subclassing. However, this declarative semantics was also used if no actual constraints are in the program. Our implementation approach in BABELSBERG/R uses different execution contexts for constraint construction/solving and imperative code. Combined with a state of the art JIT, this gives performance for pure OO code that is generally comparable to that of a standard OO VM.

7 Conclusion and Future Work

We have presented BABELSBERG, an object constraint language that extends an object-oriented language to support constraints, along with an implementation as an extension to Ruby using a state of the art virtual machine. In contrast to other approaches, BABELSBERG unifies the constructs for encapsulation and abstraction for both the declarative constraint parts of the language and the traditional imperative parts by using only object-oriented method definitions for both declarative and imperative code. Our implementation is integrated with an existing object-oriented virtual machine, and provides a standard imperative evaluation mode, a constraint evaluation mode that accumulates constraints to send to the solver as expressions are evaluated, and a constraint solver mode.

This work is recent and there are a number of directions for future research. One is to exercise the system on a wider variety of programs, and also to work on improving the performance of the constraint evaluation and satisfaction. Another direction is to add additional solvers to the library that support constraints on other primitive storage types such as arrays, strings, and hashes, and to implement a design for cooperating solvers. The multi-assignment semantics described in Section 4.4 provides a clean and simple way to control when constraint satisfaction is invoked, but experience with writing programs in Babelsberg/R is needed to decide whether this construct is sufficiently expressive. Yet another direction is to introduce a "meta-solver" that can automatically select one or more applicable solvers for a given set of constraints.

Another important focus will be adding better support for debugging, explanation, and benchmarking. (If the constraint solver is unable to satisfy the constraints, why is this? Or if the solver produces an unexpected answer, how was this answer arrived at, and are there other possible answers? If the solver is slow, why, and are there ways to make it faster? Is there a more appropriate solver available?)

Our initial implementation extends Ruby, but the ideas are applicable to other dynamic object-oriented languages, and a second implementation in JavaScript is now available [FBH+14b].

References

[BBS01] Greg J Badros, Alan Borning, and Peter J Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, December 2001. doi:10.1145/504704.504705.

[BCF+11] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'11)*. ACM, ACM, July 2011. doi:10.1145/2069172.2069181.

[BFBW92] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992. doi:10.1007/BF01807506.

[Bor12] Alan Borning. Architectures for cooperating constraint solvers. Technical Report M-2012-003, Viewpoints Research Institute, May 2012. URL: http://www.vpri.org/pdf/m2012003_coopsolv.pdf.

[BT13] Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, February 2013. doi:10.1016/j.scico.2013.02.001.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA'89)*, pages 49–70. ACM, September 1989. doi:10.1145/74877.74884.

[DGJ04] Maja D'Hondt, Kris Gybels, and Viviane Jonckers. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'04)*, pages 1328–1335. ACM, March 2004. doi:10.1145/967900.968168.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, pages 337–340. Springer, March 2008. `doi:10.1007/978-3-540-78800-3_24`.

[DR06] B. Demsky and M.C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Transactions on Software Engineering*, 32(12):931–951, December 2006. `doi:10.1109/TSE.2006.122`.

[Dud24] Henry Dudeney. Send more money. *Strand Magazine*, 214:68–97, 1924.

[Ent14] Enthought Inc. Enaml 0.6.3 documentation, February 2014. URL: `http://docs.enthought.com/enaml/`.

[FBB92a] Bjorn Freeman-Benson and Alan Borning. The design and implementation of Kaleidoscope'90, a constraint imperative programming language. In *Proceedings of the IEEE Computer Society 1992 International Conference on Computer Languages*, pages 174–180. IEEE, April 1992. `doi:10.1109/ICCL.1992.185480`.

[FBB92b] Bjorn Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. In *Proceedings of the 1992 European Conference on Object-Oriented Programming (ECOOP'92)*, pages 268–286. Springer, June 1992. `doi:10.1007/BFb0053042`.

[FBH14a] Tim Felgentreff, Alan Borning, and Robert Hirschfeld. Babelsberg: Specifying and solving constraints on object behavior. Technical Report 81, Hasso Plattner Institute, University of Potsdam, Potsdam, Germany, May 2014. Also published as TR-2013-001, Viewpoints Research Institute, Los Angeles, CA.

[FBH$^+$14b] Tim Felgentreff, Alan Borning, Robert Hirschfeld, Jens Lincke, Yoshiki Ohshima, Bert Freudenberg, and Robert Krahn. Babelsberg/JS: A browser-based implementation of an object constraint language. In *Proceedings of the 2014 European Conference on Object-Oriented Programming*. Springer, July 2014. `doi:10.1007/978-3-662-44202-9_17`.

[FBM89] Bjorn Freeman-Benson and John Maloney. The DeltaBlue algorithm: An incremental constraint hierarchy solver. In *Proceedings of the 8th Annual IEEE Phoenix Conference on Computers and Communications*, pages 538–542. IEEE, March 1989. `doi:10.1109/PCCC.1989.37442`.

[FM08] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly, January 2008.

[FMB14] Tim Felgentreff, Todd Millstein, and Alan Borning. Developing a formal semantics for Babelsberg: A step-by-step approach. Technical Report TR2014002, Viewpoints Research Institute, Los Angeles, California, July 2014.

[GF14] Alex Gaynor and Tim Felgentreff. Topaz Ruby, February 2014. URL: `http://www.topazruby.com/`.

[GH04] Martin Grabmüller and Petra Hofstedt. Turtle: A constraint imperative programming language. In *Research and Development* *in Intelligent Systems XX*, pages 185–198. Springer, 2004. doi:10.1007/978-0-85729-412-8_14.

[Hor92] Bruce Horn. Constraint patterns as a basis for object-oriented constraint programming. In *Proceedings of the 1992 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, pages 218–233. ACM, October 1992. doi:10.1145/141936.141955.

[HS96] Scott Hudson and Ian Smith. Ultra-lightweight constraints. In *Proceedings of the 1996 ACM Symposium on User Interface Software and Technology (UIST'96)*, pages 147–155. ACM, November 1996. doi:10.1145/237091.237112.

[IBM14] IBM. ILOG CPLEX Optimization Studio, June 2014. URL: http://www.ibm.com/software/info/ilog.

[ILO93] ILOG, Incline Village. *Using the CPLEX callable library and CPLEX mixed integer library*, 1993.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Principles of Programming Languages Conference (POPL'87)*, pages 111–119. ACM, January 1987. doi:10.1145/41625.41635.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, July 1992.

[KKS12] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'12)*, pages 151–164. ACM, January 2012. doi:10.1145/2103621.2103675.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 1997 European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242. Springer, June 1997. doi:10.1007/BFb0053381.

[LB97] Håkon Wium Lie and Bert Bos. *Cascading style sheets: Designing for the Web.* Addison Wesley Longman, 1997.

[LFBB94a] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Constraints and object identity. In *Proceedings of the 1994 European Conference on Object-Oriented Programming (ECOOP'94)*, pages 260–279. Springer, July 1994. doi:10.1007/BFb0052187.

[LFBB94b] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Implementing constraint imperative programming languages: The Kaleidoscope'93 virtual machine. In *Proceedings of the 1994 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, pages 259–271. ACM, October 1994. doi:10.1145/191081.191118.

[LFBB94c] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Kaleidoscope: A constraint imperative programming language. In *Constraint Programming*, volume 131, pages 313–329. Springer-Verlag, 1994. NATO

Advanced Science Institute Series, Series F: Computer and System Sciences. doi:10.1007/978-3-642-85983-0_12.

[LKI+12] Jens Lincke, Robert Krahn, Dan Ingalls, Marko Roder, and Robert Hirschfeld. The Lively PartsBin–a cloud-based repository for collaborative development of active web content. In *2012 45th Hawaii International Conference on System Science (HICSS'12)*, pages 693–701. IEEE, January 2012. doi:10.1109/HICSS.2012.42.

[LW08] Christof Lutteroth and Gerald Weber. End-user GUI customization. In *Proceedings of the 9th ACM SIGCHI New Zealand Chapter's International Conference on Human-Computer Interaction: Design Centered HCI (CHINZ'08)*, pages 1–8. ACM, July 2008. doi:10.1145/1496976.1496977.

[Mah87] Michael J. Maher. Logic semantics for a class of committed-choice programs. In *Proceedings of the 4th International Conference on Logic Programming (ICLP)*, pages 858–876, May 1987.

[MGD+90] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *Computer*, 23(11):71–85, November 1990. doi:10.1109/2.60882.

[OLFK13] Yoshiki Ohshima, Aran Lunzer, Bert Freudenberg, and Ted Kaehler. KScript and KSWorld: A time-aware and mostly declarative language and interactive GUI framework. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! '13, pages 117–134, New York, 2013. ACM. doi:10.1145/2509578.2509590.

[Pug94] Jean-François Puget. A C++ implementation of CLP. Technical report, ILOG, 1994. doi:10.1.1.15.9273.

[RCN05] Martin Rinard, Cristian Cadar, and Huu Hai Nguyen. Exploring the acceptability envelope. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 21–30. ACM, October 2005. doi:10.1145/1094855.1094866.

[RMMH+09] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, November 2009. doi:10.1145/1592761.1592779.

[RP97] Pierre Roy and François Pachet. Reifying constraint satisfaction in Smalltalk. *Journal of Object-Oriented Programming*, 10(4):43–51, August 1997.

[RP06] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA'06)*, pages 944–953. ACM, October 2006. doi:10.1145/1176617.1176753.

[Sad13] Erica Sadun. *iOS Auto Layout Demystified.* Addison-Wesley, October 2013.

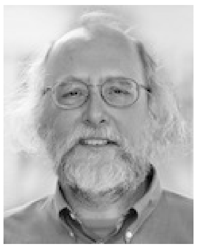
[SAM10] Hesam Samimi, Ei Darli Aung, and Todd Millstein. Falling back on executable specifications. In *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 552–576. Springer, June 2010. doi:10.1007/978-3-642-14107-2_26.

[SR93] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'90)*, pages 232–245. ACM, December 1993. doi:10.1145/96709.96733.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424, pages 632–647. Springer, April 2007. doi:10.1007/978-3-540-71209-1_49.

About the authors

Tim Felgentreff is a PhD student at the Hasso-Plattner- Institute, University of Potsdam. His research interests are in programming languages and virtual machine design. Contact him at tim.felgentreff@hpi.uni-potsdam.de. See also http://www.hpi.uni-potsdam.de/hirschfeld/people/felgentreff.



Alan Borning is a faculty member in the Department of Computer Science & Engineering at the University of Washington, at the rank of Professor since 1993, and an Associate at Viewpoints Research Institute. His research interests are in human-computer interaction and designing for human values, and in object-oriented and constraint based programming languages. He received a BA in mathematics from Reed College in 1971, and a PhD in computer science from Stanford University in 1979. Awards include a Fulbright Senior Scholar Award for lecturing and research in Australia, and being named a Fellow of the Association for Computing Machinery in 2001. Contact him at borning@cs.washington.edu.

Robert Hirschfeld is a Professor of Computer Science at the Hasso-Plattner-Institute at the University of Potsdam. There he founded and leads the Software Architecture Group which is concerned with fundamental elements and structures of software, methods, and tools for improving the comprehension and design of complex and interesting systems. He received a Ph.D. in Computer Science from the Technical University of Ilmenau, Germany. Contact him at hirschfeld@hpi.uni-potsdam.de. See also http://www.swa.hpi.uni-potsdam.de/.

Acknowledgments We would like to thank all of the members of the Software Architecture Group at Hasso Plattner Institute and of Viewpoints Research Institute for comments and suggestions on the work, in particular Bastian Steinert at HPI, and Yoshiki Ohshima and Hesam Samimi at VPRI. Alan Borning's visit to HPI was funded in part by Viewpoints Research Institute and the Communications Design Group at SAP Labs.