Babelsberg: Specifying and Solving Constraints on Object Behavior

Tim Felgentreff, Alan Borning, Robert Hirschfeld

Technische Berichte Nr. 81

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

VPRI Technical Report TR-2013-001

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam | 81

VPRI Technical Report TR-2013-001

Tim Felgentreff | Alan Borning | Robert Hirschfeld

Babelsberg

Specifying and Solving Constraints on Object Behavior

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2013
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2013/6729/
URN urn:nbn:de:kobv:517-opus-67296
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-67296

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-265-0

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 standard 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 standard object-oriented 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 generic 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.

Contents

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* in these domains is a relation that should hold, for example: 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, or that the height of a bar in a bar chart correspond to a value produced by a simulation. 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. (For example, suppose we have a constraint that $x^n + y^n = z^n$ for integers x, y, and z, with $n > 2$. Waiting 358 years for the solver to determine that the constraint is unsatisfiable is problematic. Or as another example, a language that allows the programmer to state the Post Correspondence Problem as a constraint would thus allow undecidable constraints to be specified.) On the other hand, for some restricted but very useful classes of constraints, quite efficient solvers are available. Therefore, practical solvers solvers impose restrictions on the kinds of constraints they can solve, for example, by supporting only floats and not integers or requiring numeric equalities to be linear.

So, 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 useful to support soft as well as hard constraints, that is, constraints that should be satisfied if possible

and 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. Interactive graphical systems are an example of all of these: they include constraints on complex objects such as points and areas as well as floats and integers, soft constraints to express preferences regarding the layout, and require incremental solvers to efficiently re-solve constraints repeatedly as a part of the figure is moved.

There are two main options how to use constraints in imperative programs. The most common way is via a constraint satisfaction library. This means that no changes are required to the underlying programming language, but has the disadvantage that, besides writing the constraints, 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) embedded in 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, of course at 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 standard object-oriented language that preserves a familiar imperative programming model, called BABELSBERG, and describe an implementation of that design as an extension to Ruby [13], 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 a standard object-oriented model (with dynamic typing, object encapsulation, and standard classes, instances, 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.

- The syntax of Babelsberg is a strict superset of that of the base language, with only one minor extension, making it easy for the programmer to write and read constraints and object-oriented code.
- Babelsberg includes constructs that allow specifying soft as well as hard constraints, and 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 with constraints obtain the 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 for constraint solvers that supports multiple constraint solvers, which makes it straightforward to add new solvers, and that 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.

2 Background and Related Work

Programs frequently have some set of constraints that should hold. For standard imperative languages, the usual approach to dealing with such 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 [31].

Consider a rectangle implemented as a pair of points, as in the following code. The rectangle is displayed in an application window which the user can resize. Suppose this rectangle encompasses some information that we want to make sure remains visible. We want to make sure the area of the rectangle is never less than 100 square pixels and that its origin is always within display bounds, i.e., non-negative. (It would be straightforward to include two additional tests that corner.x<=DISPLAYWIDTH and corner.y<=DISPLAYHEIGHT; these are omitted for simplicity since the visible? method already demonstrates conjunctive constraints.)

```
class Rectangle
  attr_accessor :origin, :extent

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

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

Imperatively, we can use, for example, aspects to satisfy these constraints explicitly whenever the rectangle changes:

```
class RectAspect < Aspect
  def ensure_constraints(method, rect, status, *args)
    rect.origin.x = 0 if rect.origin.x < 0
    rect.origin.y = 0 if rect.origin.y < 0
    rect.extent.x = 100.0 / rect.extent.y if rect.area < 100
  end
end

RectAspect.new.wrap(Rectangle, :postAdvice, /(origin|extent)=/)
```

However, the above code has a number of problems:

- The original constraints are expressed in a form that makes them harder to understand.
- Advice code is required to capture all modifications that may invalidate the constraints — if the advice is insufficient or not executed at the correct times, the constraints may be violated through parts of the execution.
- There are actually multiple possible solutions to the constraints, but which one should be selected is not represented explicitly, but instead is implicit in the code. (For example, a rectangle with area 200 instead of 100 would also satisfy the minimum area constraint.) Since there isn't a declarative specification of what an optimal solution is, it is nontrivial to decide whether the solution is optimal. In the presence of competing soft constraints, i.e., multiobjective optimizations, this becomes even harder.

2.1 Constraint Satisfaction Libraries

Thus, it is usually clearer to express and satisfy the constraints explicitly. One approach is to use a library that provides one or more constraint solvers, and that can be called directly from the imperative code. There is a huge range of such solvers. One way of classifying solvers is by the type domain of the constraints (for example, real numbers or finite domains or arbitrary objects); whether the solver can use the constraints in a general way (e.g., for $c = a + b$, to solve for any of a, b, or c; or as a dataflow constraint, i.e., to only use the constraint to find a value for c); and whether the solver supports soft constraints as well as hard (required) ones.

A few solvers of particular note for the programming language community are Z3 [8], a state-of-the-art SMT solver from Microsoft Research designed

for theorem proving (e.g., for program verification), and kodkod [35] for constraints over finite domains. Solvers for use in interactive graphics systems include Cassowary [2], an incremental solver for linear equality and inequality constraints that supports soft constraints as well as hard ones, the Aukland Layout Editor [26], which includes support for a GUI builder using constraints, and earlier work on DeltaBlue [16], 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 [28].

For example, the following code rewrites the previous, purely imperative solution to use the Z3 constraint solver to solve our constraints:

```
class RectAspect < Aspect
  def ensure_constraints(method, rect, status, *args)
    ctx = Z3::Context.new
    ctx << Z3::Variable.new("extent_x", rect.extent.x)
    ctx << Z3::Variable.new("extent_y", rect.extent.y)
    ctx << Z3::Constraint.new("extent_x * extent_y >= 100")
    # ... same for origin constraint
    ctx.solve
    rect.extent.x = ctx["extent_x"]
    rect.extent.y = ctx["extent_y"]
  end
end
# boilerplate code as for purely imperative approach
```

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 such as user interface layout, constraints are sometimes available via a separate DSL that describes relations between visible objects that can be automatically maintained by the runtime. Examples of such DSLs are CSS [21], the Mac OS X [1] layout specification language (which uses Cassowary to solve the constraints), and the Python GUI framework *Enaml* [11]. 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
        ]
```

This approach allows programmers to specify constraints and avoid boilerplate code to trigger constraint solving, and has 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 [29], LivelyKernel/Webwerkstatt [22], and KScript [27].

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. Programmers can thus directly express relations between objects.

```
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 this support with the programming language itself. Again, there is a substantial body of prior work in this area.

One of the most widely known of these approaches evolved from logic programming, for example the Constraint Logic Programming scheme [19] and instances of this scheme such as CLP($\mathcal{R}$) [20] (which provides constraints over real numbers). The CLP languages are in the logic programming family, and in their standard form have no notion of state or state change. Other languages of this kind include Concurrent Constraint Programming [34].

Such languages have significant advantages, such as a clean semantics, but at the same time sacrifice familiar capabilities and programming style. Our goal here is to support a more standard object-oriented, imperative programming style and syntactic integration of constraint and imperative programming. With these goals, Babelsberg follows the work by Freeman-Benson, Lopaz, and Borning on constraint imperative programming (cip) [15, 23, 24, 25] and the Kaleidoscope language. Systems related to Kaleidoscope include Siri [18], Turtle [17], and Soul [10]. BackTalk [32] is another system that aims to integrate a rich set of constraint solvers with imperative languages, but without syntactic integration.

There is also a body of work that uses constraints in other ways in standard programming languages. For example, in Plan B, Samimi, Aung, and Millstein [33] 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 [9] 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.

Kaleidoscope supported standard classes and instances, and in addition, integrated constraints with the language itself. To support this, it 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 then expanded this into constraints on the x and y instance variables of the three points a, b, and c. Separately, the language also provided methods. Both constraint constructors and standard 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.

In contrast to Kaleidoscope, Babelsberg provides a simpler semantic model, directly extending a standard object-oriented model with ordinary methods using standard object-oriented 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.

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 [14], 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) [23, 24] 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 languages. Kaleidoscope'93 also made object identity a part of

the language semantics, including support for identity constraints as well as equality constraints.

The rest of this paper is structured as follows. In chapter 3, we present the features of the language; how these features are implemented is described in chapter 4. chapter 5 presents our performance evaluation, applications written in Babelsberg/R, and a comparison of Babelsberg's features with related approaches. chapter 6 describes future work and concludes.

3 Object Constraint Programming

BABELSBERG is an object constraint programming (OCP) language — the term *object constraint programming* is chosen to emphasize the integration with standard object-oriented programming ideas, in particular methods, messages, and object encapsulation.

Our goals for BABELSBERG include a syntax that is compatible with the base language. In BABELSBERG/R, the base language is Ruby, and the extensions are almost all semantic extensions, with only one minor syntactic extension. 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).

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

```
class TemperatureConverter
  attr_accessor :centigrade, :fahrenheit
  def initialize
    # set the temperatures to well—known values
    @centigrade = 100.0
    @fahrenheit = 212.0
    always { centigrade * 1.8 == fahrenheit — 32.0 }
    # constraint added and solver triggered after block
  end
end
```

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 = 0.0 # triggers solver
# now t.fahrenheit will be 32.0
```

The constraint is labeled as **always**, indicating that it should hold indefinitely. We need to 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 — for example if we had set both of them to 0.0, one or the other would have been changed to keep the constraint satisfied. (We didn't specify which — if it mattered, this could be specified as well.)

There may be multiple constraints that together specify the desired solution. For example, here is a constraint from a Resistor class that represents Ohm's Law:

```
always { lead1.voltage − lead2.voltage == resistance*lead1.current }
```

In combination with other constraints on the behavior of batteries, wires, and grounds, and Kirchhoff's Law constraints for the connections among them, these serve to specify the behavior of electrical circuits such as a Wheatstone bridge. In this case, finding a solution involves solving a set of linear equations. (See subsection 6.1 in the appendix for a listing of these classes, along with example circuits.)

In the TemperatureConverter example, the constraints are on the results of sending the messages centigrade and fahrenheit. Here these are just accessors to the corresponding fields. However, constraints can also include methods that perform other computations. For example, the rectangle from chapter 2 exposes its area as a computed property via the area method. In Babelsberg/R, we can use existing Ruby methods from the Rectangle class to specify constraints:

```
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. 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

[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 that specify that, if it is necessary to change the value of a variable to satisfy other constraints, it should be changed as little as possible. These are left implicit in this example, but can also be stated explicitly if desired. See Section 3.6.

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) }
```

3.1 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, conjunctions and disjunctions in the method give rise to conjunctions and disjunctions in the constraints handed 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.

Second, the constraint expression should either be free of side effects, or if there are side effects, they should be benign (for example, doing caching). Also, repeatedly evaluating the block should return the same thing (so an expression involving a random number generator 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 single-assignment fashion. As an example of why this restriction is needed, consider the following method:

```
def bad_method(x)
  sum = x
  sum = sum+2
  sum
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 (which is unsatisfiable).

Babelsberg tracks assignments during constraints, so instead of a general error, it can report if a variable is assigned multiple times in a constraint. A way to address this issue would be treat multiply assigned variables in constraints as distinct variables (e.g., x_t, x_{t+1}, etc.), which are connected through equality constraints, similar to the original Kaleidoscope'90 refinement approach. However, we believe that the additional complexity of keeping track of successive versions of a variable outweighs the additional expressiveness, and have not supported such a feature. (Note that for straight-line code it's easy to add additional variables to satisfy this restriction. Loops with such multiply assigned variables need to be converted to recursions — see the following section for an example.)

3.2 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, just as with any other statement. 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]

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

Similar constructs can be used in representing more complicated structures. For example, the Kirchhoff's Law constraint in the electrical circuit examples listed in subsection 6.1 uses inject to constrain the sum of the currents in n connected leads to be 0.

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:

[2] The sum method makes of the inject method instead of, for example, a call to each that repeatedly assigns to a local variable, so that we satisfy Babelsberg's single assignment rule for variables used in constraints. Because each block activation creates a separate scope, each partial sum is a separate variable.

```
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`:

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

3.3 Solvers

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 array elements that maps to Z3 and Cassowary, and DeltaBlue.

Currently, the programmer must explicitly indicate which solvers are available for a given program. The solver for a given constraint is chosen eagerly, based on the run-time type of the variables that occur in the constraint, and 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), the programmer has to decide which solver to use by activating only the desired one. In some cases, this choice may simply be a matter of preference, however, other restrictions apply. For example, Cassowary cannot solve nonlinear equations. To solve nonlinear constraints, the developer has to know about this restriction and choose Z3. For interactive graphical applications where constraint priorities are needed, Cassowary has to be used.

Developers can 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 repeatedly 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 the constraints. See section 2.

3.4 Read-only Variables

Babelsberg includes support for read-only variables, expressed by sending the question-mark method (*?*) to a value in a constraint expression. The implementation of this method asserts an additional equality constraint in the active solver. A read-only variable can only be changed by other solvers upstream of the constraint with the read-only variable, or imperatively, but not to satisfy the constraint in which it occurs. (See reference [6] for a formal definition of read-only variables. While the formal definition allows cycles through read-only variables, in practice we haven't found use cases for this, and it is confusing for programmers, so in Babelsberg we disallow this.)

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 could choose to change the local variable desiredsize rather than the receiver.

Read-only variables also play a central role in our architecture for cooperating constraint solvers, serving as "gateways" to control information flow among the solvers. See section 2.

3.5 Constraint Durations and Activation

Constraints have *durations* during which they are active. So far, we've been using the **always** duration, which declares that its constraint becomes active when the **always** statement is evaluated and remains active indefinitely after that. A **once** constraint is activated, satisfied, and then retracted. Finally, an assert−during constraint is active for the duration of the evaluation of its associated block.

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, all values are assigned before the solver is triggered.

3.6 Soft Constraints and Stay Constraints

The concept of a constraint has been extended to include soft constraints as well as hard (required) ones, and a number of constraint solvers, including Cassowary and DeltaBlue, support them. Soft constraints are useful, for example, in interactive graphical applications to represent a requirement that two icons be separated by a minimum of 10 pixels (a required constraint) and the desire that they be 50 pixels apart if possible (a soft constraint). Soft constraints can have different priorities, with the higher-priority constraints satisfied in preference to lower-priority ones. There are a variety of ways of trading off two soft constraints of the same priority that cannot both be satisfied; see [6] for details.

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}
```

Only some constraint solvers support soft constraints, so whether soft constraints can be handled depends on which solver is in use. Also, in Babelsberg the choice of how soft constraints are traded off is a property of the selected solver.

An important application of soft constraints is to express the desire that parts of a system should remain the same unless there is some reason they should change (for example to satisfy some higher priority constraints). This is a central issue when combining constraints with imperative programming, since in such a language we need to specify 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.

3.7 Incrementally Re-satisfying Constraints

Some applications involve repeatedly re-satisfying the same set of constraints with differing input values. A common such 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, and a number of constraint solvers support this using so-called *edit constraints* that allow a new value for a variable to be repeatedly input to the solver. Babelsberg also supports incrementally re-satisfying constraints with an edit method that takes a variable to be edited, a stream (that responds to **next**) that provides the new values, and optionally a priority. 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:

```
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. Babelsberg/R takes advantage of incremental solvers when available, so that the constraints can be re-satisfied very quickly. The implementation of edit constraints is discussed in section 4.3.

3.8 Constraints on Object Identity, Class, and Message Protocol

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

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

If we are only using such constraints as tests, it is straightforward to check them. Satisfying identity constraints is straightforward, but should be handled with a specialized solver. In our current implementation, identity constraints are supported through the provided DeltaBlue library. 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 nice benefit of this design is that type declarations or their equivalent are just constraints.

4 Implementation

We first provide an overview of the key features of the implementation before plunging into the details. The interpreter normally operates in imperative evaluation mode. In the absence of constraints, this is a standard Ruby virtual machine (VM). However, if the interpreter encounters a STORE instruction for a variable with a constraint on it, rather than directly storing into the variable, it calls the appropriate constraint solver to solve an equality constraint between the variable and the new value; and if it encounters a LOAD instruction for a constrained variable with a constraint, it gets the value from the solver for that variable. Constraints in BABELSBERG/R are written as ordinary Ruby expressions that return true or false depending on whether the constraint is satisfied. When a constraint is being added (using for example **always**), the interpreter switches to constraint construction mode. It continues to evaluate expressions using message sends, but rather than computing the result, it instead builds up a network of primitive constraints that represent the constraint being added. In the process, it keeps track of all the dependencies. If later a variable is assigned to that is involved in that constraint, its solver will be activated, or if there is none, the constraint expression will be re-evaluated (in constraint construction mode) to update the stored constraint network.

Going now into more detail, the Ruby VM we use as a basis for BABELSBERG/R is *Topaz* [12], an experimental VM built using the *PyPy/RPython* toolchain [30]. This has allowed us to extend the interpreter and use RPython's VM-generation toolchain to create a VM including a fast JIT and garbage collector.

The changes we made to the Topaz VM are two-fold. First, we added an interpreter mode for constraint construction (cf. section 4.2) and a primitive to enter this mode. Second, we extended the interpreter to support constrained variables (cf. section 4.1) by allowing the same name to refer to multiple objects, one for use in imperative evaluation mode and the other for constraint construction mode. (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.)

4.1 Constrained Variables

Ruby provides 5 types of variables: locals, instance variables, class variables, globals, and constants. (While constants are supposed to be assigned only once, it is not an error condition to re-assign a constant in Ruby.) Of these, we allow 3 as constrained variables: locals, instance variables, and class variables.

Variables are constrained by using them in a constraint expression. In Topaz, all locals are stored in closure cells and we simply had to add a second field for constraint values. Instance and class variables are stored in an array and Topaz uses map transitions [7] to access them efficiently. For Babelsberg/R, have added a transition to store constraint values as well as object-oriented (OO) values in these maps.

4.2 Execution Contexts

To implement the different execution contexts — *imperative execution*, *constraint construction*, and *constraint solving* — we added ConstraintInterpreter as a subclass of Interpreter that changes how locals, instance variables, and class variables are accessed. We also changed the default interpreter to be aware of changes from solvers.

Imperative Execution

During imperative execution, the LOAD bytecodes check whether a variable also refers to a constraint value and if so, copy the value the solver has determined to the OO value. For immutable objects (such as small integers), the VM replaces the reference to the OO value and returns the new object. For mutable objects, the VM calls assign_constraint_value with the solver-supplied value. The object must then adjust itself, before it is returned.

STORE instructions similarly check whether the assigned variable has a constraint value. There are two situations in which this will be the case. In the first, the variable itself is in use by an active solver. In that case, the interpreter, instead of storing the supplied value directly, calls the suggest_value method on the constraint value, with the new value as argument. This will in turn invoke the associated solver. In the second, the variable is on the path of a constraint, but is itself not used by a solver. In this case, all constraint blocks that encountered this value during their execution must be invalidated and re-executed with the new value.

As described in section 3.5, in Babelsberg/R, if multiple variables that have constraints on them are assigned using multi-assignment, all values are assigned before the solver is triggered.

Constraint Construction

The initializer for constraint objects — usually called through **always** — activates an execution context for constraint construction. In this mode, LOAD instructions find or create a constraint marker for each value that is accessed. These markers send for_constraint to the value, which is expected to return an object that responds to value and suggest_value (where during execution of the latter it would use a solver to re-satisfy constraints). For values that do not respond to for_constraint, the marker is left to recalculate the constraint when the value changes. Otherwise, the object returned from for_constraint is permanently associated with the variable as its constraint value (unless it becomes inaccessible and is garbage collected). The constraint value replaces the actual value for the duration of the constraint construction.

A constraint value 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 by Float#*for_constraint* 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.

STORE instructions in this mode create equality constraints. This is necessary to support constructing new objects in the predicate that connect values. However, this also means that all code blocks encountered during constraint construction have to be in single assignment form (section 3.1).

Support to pass branches to solvers that support them (such as **if** ... **then, or**, ...) is currently not available. We have added support for the conjunction bytecode only by creating two (or more) primitive constraints that are collected in the constraint object and can be enabled and disabled simultaneously.

Note that 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. We currently expect solvers to be subclasses of ConstraintObject. We use this class as a marker to leave the constraint execution context when we enter solver code.

Constraint Solving

This execution context is entered when storing into a constrained variable. The VM sends suggest_value to the ConstraintObject for this variable. This context prevents nested sends of suggest_value to avoid recursions in the solver.

Our current implementation of execution contexts uses Python's with statement to push a context onto the stack. At those points in the VM that need to be adapted, we dispatch through the context. A new context is created when a Constraint object is initialized, when a constrained variable is assigned and we need to re-satisfy constraints, and when we send a message to an object that inherits from ConstraintObject.[1]

Constraint Construction Example

To illustrate how our implementation supports the combination of objects and constraints, consider the rectangle example in BABELSBERG from chapter 3. 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 and sending the >= method to the result. Explicitly named in the constraint is only the variable rect, but there are other variables that play a role in this constraint.

A constraint in our system is a predicate expression passed unevaluated in form of a block closure to the primitive method **always**. The closure is executed in a *constraint interpreter* mode that resolves variables to constraint values instead of OO values. The constraint interpreter otherwise executes the code using normal OO semantics.

In our example above, the rect variable is replaced with a placeholder, because no rectangle-solver exists and the class Rectangle does not implement the method for_constraint. The placeholder delegates the message area to the OO value. During the execution of area a number of float values are encountered — the instance variables of the points that form the rectangle (cf. Figure 4.1). These instance variables are resolved to constraint values, for example Cassowary variables, and the messages between them, instead of calculating the current area of the rectangle, return a linear constraint expression. This linear

[1] If we wanted to support other cooperating paradigms in addition to constraint-oriented programming, we would need to extend this to support different execution contexts in a more general way.

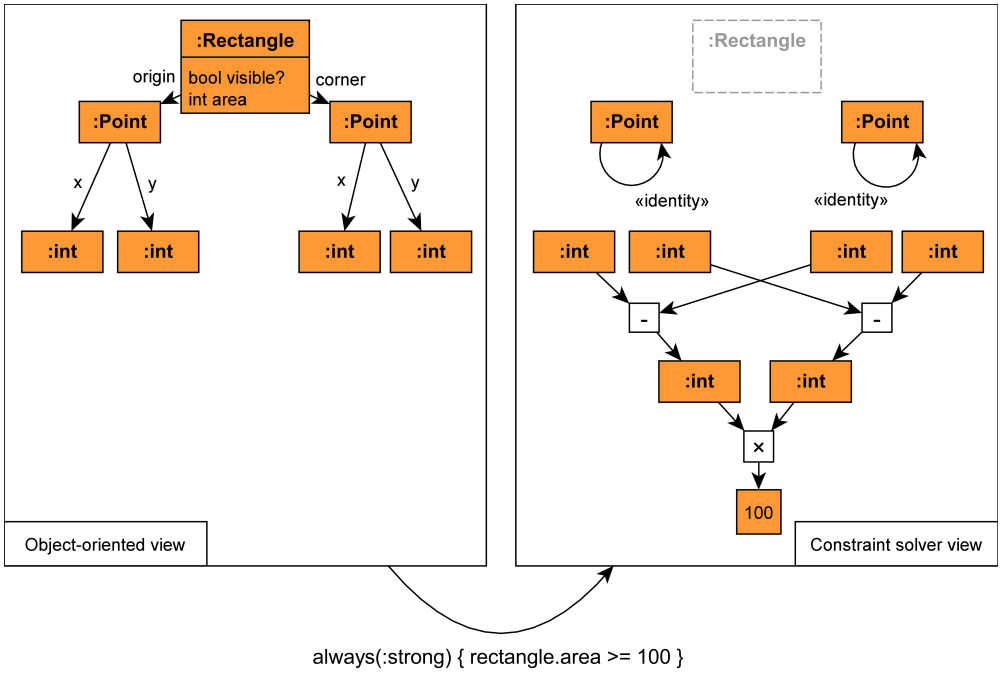

Figure 4.1: Objects are connected through instance variables. When a constraint is constructed, objects become connected through constraints as well.

expression is then sent the message >= with 100 as its argument and returns an inequality constraint that can be solved by Cassowary.

The constraint construction is complete when block passed to **always** returns. Any constraint values and combinations thereof created during the execution are gathered into a *Constraint*. These provide a meta-level interface to enable or disable the entire constraint (and thus all participating values), query and change its priority (cf. subsection 3.2), and access the block that was passed to **always**, as well as each constraint value that participates in the constraint.

The constraint then includes a) primitive values that are in the type domain of the solver and are connected in a constrainable way (e.g., through equalities or inequalities, arithmetic, boolean logic) and b) a number of marked values for complex objects that invalidate the constraint if their identity changes.

The constraint solvers operate on primitive constraint values directly to solve the constraint. Changes made by the solver to the variables show up in the OO view on the next access to the variables from imperative code, at which point the constraint 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).

Placeholder constraint values invalidate the constraint if the identity of the OO value changes through imperative assignment. This invalidation retracts all constraint values created during constraint construction and re-executes the block to create new constraint values. This means that the block may be re-executed multiple times during the run-time of the program, and any side-effects should be benign (cf. section 3.1).

4.3 Implementing Edit Constraints

Edit constraints are used to support incremental constraint satisfaction, and are important to achieve good performance in interactive applications. The edit method adds an edit constraint on the edit variables that are in the block, and then repeatedly calls the solver's suggest_value method with each new value from the stream. When the stream is empty, it removes the edit constraint and returns. The priority defaults to the highest non-required one. (If :required is used, the system raises an exception if the new edit value can't be used.) 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.

Babelsberg does not support edit constraints at a language level. Instead, the definition of the edit method is part of the Cassowary solver library that is included in the current version of Babelsberg/R. The library uses the application programming interface (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.

Cassowary as shipped in the Babelsberg/R standard library allows the variables and stream to hold user-defined objects, not just primitive types. To use Cassowary as the solver, we need to convert these into edit constraints on Floats or other primitive objects (e.g., the x and y values of the midpoint), but we also want to do this in an object-oriented way that respects encapsulation. Further, updating the solution with each new value should continue to be very fast.

To support this, the client passes an array of accessor method names for the return values that should be updated in the edit constraint. Cassowary internally creates external variables that it uses as edit variables. Those ex-

ternal variables are equality constrained to the return values of the accessor methods, so the internal storage layout of the class does not matter. 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,
     accessor: [:x, :y]) { mouse_point }
```

For DeltaBlue, the edit constraints returned could be simpler, e.g., the point would be simply updated rather than dealing with its x and y coordinates separately.

4.4 Adding new Solvers

During constraint construction, the VM sends for_constraint to each variable encountered during the execution. User code can add solvers to the system by dynamically adding a for_constraint method to those classes that the solver can work with, 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. For Cassowary, we extended 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.

We expect the objects returned from a for_constraint method to respond to two messages:

suggest_value When imperative code stores into a variable that participates in constraints, the VM instead sends this message to the associated constraint object. This method triggers the solver, raising an exception if a required constraint can no longer be satisfied.

value Whenever the VM accesses a variable that participates in a constraint, it sends this message to the associated constraint object to determine whether the variable was updated by the solver.

In the case of immutable objects (such as floats and integers), the VM updates the reference for the variable directly. However, for mutable objects, solver libraries should provide the method assign_constraint_value to update the object. We provide such a method for the Array class, which replaces the array's contents with the contents suggested by the solver:

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

This illustrates how the solvers provided by BABELSBERG/R are simply constraint solver libraries that extend core classes.

5 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.

5.1 Performance Evaluation

There are three questions we are interested in regarding performance: a) how does constraint solving performance compare to writing imperative code to satisfy constraints, b) how is object-oriented performance affected by our extensions to the VM, and c) how does a practical application perform if refactored to use constraints.

Constraint Solving

Toward answering the first question, we used an example from [25] and adapted it to Babelsberg/R. 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.

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. In our benchmarks we have run both variants.

Table 5.1 shows that the naive object-constraint version is usually many hundred times slower than the purely imperative solution. As the iteration count grows, the naive version also does not seem to benefit as much from the JIT as the imperative version. Using edit constraints, repeated solving of constraints is much faster. Keep in mind that these results use a solver that is

```
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 5.1: Interactive thermometer example from [25]

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 5.1: Thermometer benchmark (normalized against imperative version)

also written in pure Ruby, which is by itself orders of magnitude slower than a C++ based version. Considering that Babelsberg/R is aimed at imperative programmers, we expect that most of the time the VM will not be solving constraints in tight loops, but running mostly imperative code intermingled with constraint re-satisfaction.

Object-oriented Performance

For imperative code performance, we ran a number of tests from the meta-tracing VMs experiment [4] against the unmodified *Topaz* Ruby VM and the *JRuby* VM.

For purely imperative code, Babelsberg/R is consistently less 1.5 times slower in these benchmarks than the unmodified VM (Figure 5.2). The only benchmark where we are doing significantly worse than Topaz is *Binarytrees*.

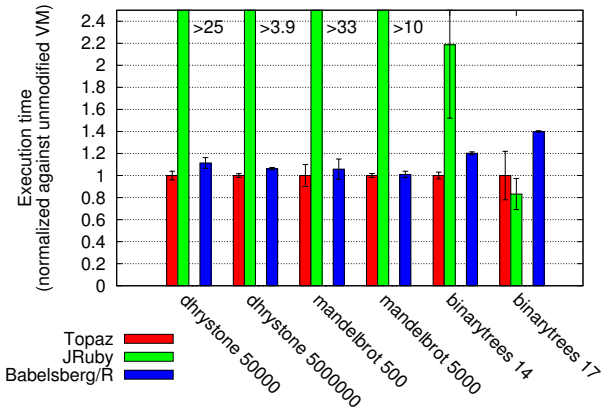

Figure 5.2: Metatracing VM benchmark run on an Intel i7 Quad-Core CPU with 3.4 GHz

Binarytrees is a strongly recursive benchmark, which Topaz' (and thus Babelsberg/R's JIT) is bad at optimizing (which is why JRuby does better in this benchmark). The overhead in Babelsberg/R is mostly due an additional test each time a variable is accessed, to check whether this variable participates in a constraint. An annotation in the VM [3] allows the JIT to remove this test when there are no constraints on variables in the loop, but in *Binarytrees*, we do not generate machine code for a large part of the code (due to its recursive nature), and the test incurs some overhead. For code where we generate a sufficient amount of machine code, though, Babelsberg/R is generally around 10% slower than Topaz (or less than 20% including standard deviation).

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 enough 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:

```
def recalculate_quality
  if @quality + encoding_time < 90
    @quality = @quality + encoding_time
  elsif @quality + encoding_time > 100 + FrameMsMax
    @quality = 100 + FrameMsMax
  end

  if cpuload > 0.8 * NumberOfProcessors
    diff = 100 * (NumberOfProcessors - cpuload)
    if diff > 0
      @quality -= diff
    else
      @quality += diff
    end
  end

  if @quality > user_preference
    @quality = user_preference
  end
  @quality = 0 if @quality < 0
  @quality = 100 if @quality > 100
end
```

(a) Imperative version

```
always { @quality >= 0 }
always { @quality <= 100 }
always { @quality <= user_preference }
always(:high) { @quality == user_preference }
always { @quality + cpuload * 100 <=
         100 + 80 * NumberOfProcessors }
always(:high) { @quality + encoding_time >= 90 }
always { @quality + encoding_time <=
         100 + FrameMsMax }
```

(b) Object-constraint version

Figure 5.3: Quality constraints in video streaming

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.

VM Hooks for Customizing Constraint Construction

The VM hook methods for_constraint and assign_constraint_value are useful not only for solvers, but also when users want to provide a particular interpretation of certain application domain objects in their constraints. In Figure 5.3 we use 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 minimal. The following class definition was enough for us to be able to use the execution time of a method in a constraint:

```
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 can be instantiated with a class and the name of a method, and wraps this method to record its last execution time. When used in a constraint, a read-only constraint object is returned that was associated 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
```

	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 5.2: Comparison of OCP with related work

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.

5.2 Comparison with Related Approaches

BABELSBERG has a number of properties that we think are desirable for an object-constraint programming language. Table 5.2 shows these properties and compares OCP to related approaches presented in chapter 2. As a continuation of CIP, OCP shares most properties with the 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. If constraints are used, they respect encapsulation and re-use the object-oriented method definitions. 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, functional reactive programming (FRP) and CIP languages use *propagation hooks* and *constraint constructors* respectively to support constraints. Here, propagation hooks are the functions connecting variable that are called when one variable changes to compute a new value for the downstream variable. Constraint constructors are the concept in Kaleidoscope to build higher order constraints. In the simplest case these are rewrite rules to convert constraints on complex objects into primitive constraints for a solver. In Kaleidoscope,

messages in constraint expressions are dispatched to constraint constructors using multi-method lookup semantics, where the receiver and all arguments participate in the method selection.

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. [10] argue that linguistic symbiosis between different programming paradigms is required to support the evolution of programs 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, the Wheatstone bridge example 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 the "Conclusion and Future Work" section, 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 a standard OO VM as used for other approaches.

6 Conclusion and Future Work

We have presented Babelsberg, an object constraint language that extends a standard 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 hand off 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 work. 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, for example a finite domain solver or solvers that support constraints on other primitive storage types such as arrays, strings, and hashes, and to implement the design for cooperating solvers. The multi-assignment semantics described in section 3.5 provides a clean and simple way to control when constraint satisfaction is invoked. However, more experience with writing programs in Babelsberg/R is needed to decide whether this construct is sufficiently expressive, or whether additional ways are needed to control when constraint solving is invoked. Yet another direction regarding solvers is to introduce a "meta-solver" that can automatically select one or more applicable solvers for the 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? 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 an implementation in Javascript is currently underway.

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 SAP Labs.

Bibliography

[1] Apple Inc. *Cocoa Auto Layout Guide*, September 2012.

[2] 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, 2001.

[3] 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*, page 9. ACM, 2011.

[4] Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, 2013.

[5] Alan Borning. Architectures for cooperating constraint solvers. Technical Report VPRI Memo M-2012-003, Viewpoints Research Institute, Glendale, California, May 2012.

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

[7] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *ACM SIGPLAN Notices*, 24(10):49–70, 1989.

[8] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[9] B. Demsky and M.C. Rinard. Goal-directed reasoning for specification-based data structure repair. *Software Engineering, IEEE Transactions on*, 32(12):931–951, 2006.

[10] M. D'Hondt, K. Gybels, and V. Jonckers. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1328–1335. ACM, 2004.

[11] Enthought Inc. Enaml 0.6.3 documentation, 2013.

[12] Tim Felgentreff. Ruby Topaz. wroc_love.rb, March 2013.

[13] David Flanagan and Yukihiro Matsumoto. *The ruby programming language*. O'Reilly, 2008.

[14] 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 International Conference on Computer Languages*, pages 174–180, April 1992.

[15] Bjorn Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 268–286, June 1992.

[16] Bjorn Freeman-Benson and John Maloney. The DeltaBlue algorithm: An incremental constraint hierarchy solver. In *Proceedings of the Eighth Annual IEEE Phoenix Conference on Computers and Communications*, Scottsdale, Arizona, March 1989. IEEE.

[17] Martin Grabmüller and Petra Hofstedt. Turtle: A constraint imperative programming language. In *Research and Development in Intelligent Systems XX*, pages 185–198. Springer, 2004.

[18] 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*, pages 218–233, Vancouver, British Columbia, October 1992.

[19] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Fourteenth ACM Principles of Programming Languages Conference*, Munich, January 1987.

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

[21] Håkon Wium Lie and Bert Bos. Cascading style sheets: Designing for the web, 1997.

[22] 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 *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 693–701. IEEE, 2012.

[23] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Constraints and object identity. In *Proceedings of the 1994 European Conference on Object-Oriented Programming*, pages 260–279, July 1994.

[24] 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*, pages 259–271, October 1994.

[25] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Kaleidoscope: A constraint imperative programming language. In Brian Mayoh, Enn Tyugu, and Jaan Penjam, editors, *Constraint Programming*. Springer-Verlag, 1994. NATO Advanced Science Institute Series, Series F: Computer and System Sciences, Vol. 131. Also published as UW CSE Technical Report 93-09-04.

[26] 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*, pages 1–8. ACM, 2008.

[27] Yoshiki Ohshima, Bert Freudenberg, Aran Lunzer, and Ted Kaehler. A report on KScript and KSWorld. *VPRI Research Note 2012-008*, 2012.

[28] CPLEX Optimization. Using the cplex callable library and cplex mixed integer library. *CPLEX Optimization, Incline Village*, 1993.

[29] 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, 2009.

[30] 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*, pages 944–953. ACM, 2006.

[31] M. Rinard, C. Cadar, and H.H. Nguyen. Exploring the acceptability envelope. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 21–30. ACM, 2005.

[32] P. Roy and F. Pachet. Reifying constraint satisfaction in Smalltalk. *JOOP*, 10(4):43–51, 1997.

[33] 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 Berlin Heidelberg, 2010.

[34] Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[35] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

Appendix

This appendix contains material not published in the journal article, including a design for additional language features not yet implemented, and example programs. All the example programs are fully functional with the current implementation.

1 About the Name

Babelsberg is a district of Potsdam, in which HPI is located. It is also the location of Studio Babelsberg, the historical center of the German film industry (Fritz Lang's *Metropolis* and Josef von Sternberg's *The Blue Angel* were filmed there), and still active, primarily for high-budget feature films.

2 Support for Multiple Cooperating Solvers

The current implementation supports multiple solvers, and a protocol for solvers so that it is convenient to add new ones. However, for a given program only one solver is used. This is a significant limitation, since richer sets of constraints can be written and solved using multiple cooperating solvers. The Viewpoints note "Architectures for Cooperating Constraint Solvers" [5] describes one technique for supporting this, which fits well with Babelsberg. Briefly, in the "Architectures for Cooperating Constraint Solvers" approach, we view the constraints and constrained variables as a bipartite graph. For the top level, we partition the constraints into regions, based on the following requirements:

- each constraint belongs to exactly one region
- the different regions can share variables (but not constraints)
- for each variable shared by two or more regions, that variable must be read-only for all but one of the regions

- the graph of the different top-level regions must be acyclic
- it isn't possible to divide any region into two regions and still maintain the other requirements

We can always partition the constraint graph this way. (In the limit, there is a single region.) The partitioning is unique.

Given this partitioning, and a solver for the constraints in each region, it is easy to solve the constraint system as a whole: it is just a dataflow problem. We find an order for solving the regions, so that region B is solved after region A if B is downstream from A in terms of the read-only variables.

3 Initializing and Manipulating Constraints

This subsection describes the protocol for initializing constraints and the methods available on constraint objects.

Formally, a constraint is just a predicate that we want to have hold. In the implementation, it is represented as an instance of Constraint. Minimally, a constraint can be defined just as a predicate, written as a Ruby block. The block should meet the following criteria:

- Evaluating the block returns a boolean — the constraint is that it evaluate to true.
- The expression in the block should either be free of side effects, or if there are side effects, they should be benign (for example, doing caching).
- Repeatedly evaluating the block should return the same thing (so an expression involving a random number generator wouldn't qualify).

There should be other available attributes for Constraint. Constraints that will be satisfied using a local propagation solver need a set of propagation methods, soft constraints need a priority, and so forth.

3.1 Initializing a Constraint

Here is a list of attributes that can be supplied when initializing a constraint object. The predicate is the only required one, and as described below can be either an argument or a block after the arguments. All of the other attributes are specifiable as optional arguments to **always** or **once**.

predicate A block that evaluates to true or false.

priority Defaults to :required.

weight Defaults to 1.

error A block that evaluates to the error in satisfying that constraint. The error value is 0 iff the constraint is satisfied. Defaults to a simple 0/1 function.

methods (optional) A dictionary of local propagation methods, indexed by the variable whose value is computed by the corresponding method.

The local propagation methods are only needed for local propagation solvers like DeltaBlue. Each method is a block, that if evaluated, will set the index variable to a value that satisfies the constraint. The constraint should be functional for the writable variables, i.e., the value for the index variable should be unique given values for the other constrained variables. If a constraint is read-only in some variable, then there isn't a corresponding block for that variable in the dictionary.

always can be used to define compound constraints. This happens, for example, if a predicate method checks two conditions and conjoins them (as, for example, in a CartesianPoint#== method). All arguments apply to all constraint parts that are created during the execution of **always**. So a :low priority compound constraint will be added to the solver as multiple primitive constraints that are all :low priority. If local propagation methods are provided, evaluating a method must satisfy all of the component constraints of a compound constraint. The error for a compound constraint is likely to be problematic unless it is a simple 0/1 function.

3.2 Methods for the Class Constraint

The Constraint class provides the following methods in Babelsberg/R. (Here we are referring to ordinary Ruby methods provided by the class, not the constraint methods for local propagation.)

enable Enable all constraints that were asserted in the block. (If the constraint is a compound constraint added to the solver as multiple primitive constraints, all of these primitive constraints are enabled.)

disable Disable all constraints that were asserted in the block.

priority Returns the constraint block priority.

priority= Setter for the constraint block priority. This retracts all constraints and re-enables them using the new priority.

constraint_block Returns the block object that was passed to **always**.

solver_constraints Returns an array with the solver-specific constraint objects. For Cassowary constraints, this is a list of LinearEqualities, for Z3 it is a list of Z3Asts.

4 Constraint Durations

A constraint may not necessarily be active for the entire life of the program. BABELSBERG provides the following options: **always**, **once**, and assert {}. during {} (for constraints that are enforced while a block is executing and then removed).

The return value of a call to **always** is an object of type Constraint. If the constraint has no solver, but the expression already evaluates to true, a warning is printed; if it is false, an exception is raised.

Note that currently all assertions that we want to continuously check, but cannot solve (yet), are implemented in this way. For example, without enabling any string solvers in the system, **always** { stringA == stringB } will return **true** if indeed the two strings are equal, and whenever any of the strings is assigned, the block is re-executed to check if the condition still holds. If it doesn't (the boolean returned is be false or nil), an error is raised.

This same mechanism is used when complex objects for which we have no solver occur in the constraint expression (even if a primitive constraint is created in the block). Consider equality between two Cartesian points, pt1 and pt2: the constraint expression will create two primitive constraints, namely that pt1.x == pt2.x and pt1.y == pt2.y. Whenever an @x or a @y of either p1 or pt2 is assigned, the constraint solver will just calculate new values for the other instance variables. However, if pt1 is assigned a new object, the two primitive constraints are removed from the solver and the constraint block is executed again, to find the new meaning of pt1.x == pt2.x. Because pt1 may now be a different class, the equality may be expressed differently. So whenever a complex object occurs in the execution of a constraint expression, the entire block is kept alive as long as the object is, so that the block can be re-executed later when the object is assigned a new value.

5 Constraint Priorities

All arguments to **always** (except for the block), are simply passed on to the enable method of the solver specific primitive constraint objects asserted during the execution of the block. BABELSBERG/R does not know about constraint priorities (or weights, local propagation methods, or anything else that a particular solver may use). DeltaBlue and Cassowary—the two solvers that we currently provide that support priorities—simply map a set of named priorities to internal priority values: required, high, medium, and low. We currently require that all the solver libraries use the same priority names. (Adding a more general mechanism that allows arbitrarily many priorities, and different names in different libraries, is a possible extension. However, in practice we've found that the fixed set of priorities has been sufficient, because the priorities tend to be used in stylized ways, e.g., high for edit constraints and low for stays.)

6 Example Programs

This part of the appendix provides a set of example programs in BABELSBERG/R that illustrate various features of the language

6.1 Electrical Circuit Examples

Here is a complete listing of the classes for building simple electrical circuits with batteries, resistors, wires, and grounds, along with some sample circuits using them. Note in particular the clean way constraints such as Ohm's Law can be expressed, and the integration of constraints with control structures for the Kirchhoff's Law constraint on n connected leads that their currents sum to 0.

```
class Lead
  attr_reader :voltage, :current
  def initialize
    # set voltage and current to 0.0 for now so that they are of type Float
    # (constraints may change them later)
    @voltage = 0.0
    @current = 0.0
  end
end

class TwoLeadedObject
  attr_reader :lead1, :lead2
```

```
  def initialize
    @lead1 = Lead.new
    @lead2 = Lead.new
    # constrain currents to be equal magnitude and opposite
    always { lead1.current + lead2.current == 0.0 }
  end
end

class Resistor < TwoLeadedObject
  attr_reader :resistance
  def initialize (resistance)
    super()
    @resistance = resistance
    # Ohm's Law constraint. Note that the resistance is read—only.
    always { lead1.voltage — lead2.voltage == resistance.?*lead1.current }
  end
end

class Battery < TwoLeadedObject
  attr_reader :supply_voltage
  def initialize (supply_voltage)
    super()
    @supply_voltage = supply_voltage
    always { lead2.voltage — lead1.voltage == @supply_voltage.? }
  end
end

class Ground
  attr_reader :lead
  def initialize
    @lead = Lead.new
    # constrain the voltage and current to be 0
    always { lead.voltage == 0.0 }
    always { lead.current == 0.0 }
  end
end

class Wire < TwoLeadedObject
  def initialize
    super()
    always { lead1.voltage == lead2.voltage }
  end
end

def connect(*leads)
  return if leads.empty?
  # all voltages should be equal
  leads[1..—1].each { |a| always { a.voltage == leads[0].voltage } }
  # sum of currents has to be 0
  always { leads.map(\&:current).sum == 0 }
end

def battery_resistor
  # build a circuit with a battery and a resistor , and return the current in one of the leads
```

```
  r = Resistor.new(resistance)
  b = Battery.new(voltage)
  connect g.lead, r.lead1, b.lead1
  connect r.lead2, b.lead2
  return r.lead1.current
end

def wheatstone_bridge
  # build a Wheatstone bridge, and return the current through the
  # galvanometer (represented as a wire in this version)
  g = Ground.new
  b = Battery.new(5.0)
  r1 = Resistor.new(100.0)
  r2 = Resistor.new(100.0)
  r3 = Resistor.new(100.0)
  r4 = Resistor.new(100.0)
  w = Wire.new
  connect g.lead, b.lead1, r3.lead2, r4.lead2
  connect b.lead2, r1.lead1, r2.lead1
  connect r1.lead2, r3.lead1, w.lead1
  connect r2.lead2, r4.lead1, w.lead2
  return w.lead1.current
end
```

6.2 Cryptarithmetic

Here is the well known cryptarithmetic puzzle send+more=money, which BABELSBERG/R can solve using Z3. (Later, we should also have an efficient finite-domain solver such as kodkod available.)

```
# initialize each variable to an integer so that the solver knows its type
s,e,n,d,m,o,r,y = [0]*8

# each letter represents a digit between 0 and 9
always { [s,e,n,d,m,o,r,y].ins(0..9) }

# each letter is mapped to a different digit
always { [s,e,n,d,m,o,r,y].alldifferent ? }

# the addition constraint
always {   s*1000 + e*100 + n*10 + d +
           m*1000 + 0*100 + r*10 + e ==
 m*10000 + 0*1000 + n*100 + e*10 + y }

# the leading digits can't be 0
always { s>0 }
always { m>0 }

puts ("solution: [s,e,n,d,m,o,r,y] = " + [s,e,n,d,m,o,r,y].to_s)
```

Colophon

This report was typeset by LaTeX 2ε with pdfTeX using KOMA-Script. The body text is set 11/14¼ pt on a 30¼ pc measure. The body type face is Hermann Zapf's *Palatino Linotype*. The listing type face is *DejaVu Sans Mono*, based on the *Vera* family by Bitstream, Inc. —*Tobias Pape*

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
80	978-3-86956-264-3	**openHPI: The MOOC Offer at Hasso Plattner Institute**	Christoph Meinel, Christian Willems
79	978-3-86956-259-9	**openHPI: Das MOOC-Angebot des Hasso-Plattner-Instituts**	Christoph Meinel, Christian Willems
78	978-3-86956-258-2	**Repairing Event Logs Using Stochastic Process Models**	Andreas Rogge-Solti, Ronny S. Mans, Wil M. P. van der Aalst, Mathias Weske
77	978-3-86956-257-5	**Business Process Architectures with Multiplicities: Transformation and Correctness**	Rami-Habib Eid-Sabbagh, Marcin Hewelt, Mathias Weske
76	978-3-86956-256-8	**Proceedings of the 6th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering**	Hrsg. von den Professoren des HPI
75	978-3-86956-246-9	**Modeling and Verifying Dynamic Evolving Service-Oriented Architectures**	Holger Giese, Basil Becker
74	978-3-86956-245-2	**Modeling and Enacting Complex Data Dependencies in Business Processes**	Andreas Meyer, Luise Pufahl, Dirk Fahland, Mathias Weske
73	978-3-86956-241-4	**Enriching Raw Events to Enable Process Intelligence**	Nico Herzberg, Mathias Weske
72	978-3-86956-232-2	**Explorative Authoring of ActiveWeb Content in a Mobile Environment**	Conrad Calmez, Hubert Hesse, Benjamin Siegmund, Sebastian Stamm, Astrid Thomschke, Robert Hirschfeld, Dan Ingalls, Jens Lincke
71	978-3-86956-231-5	**Vereinfachung der Entwicklung von Geschäftsanwendungen durch Konsolidierung von Programmier-konzepten und -technologien**	Lenoi Berov, Johannes Henning, Toni Mattis, Patrick Rein, Robin Schreiber, Eric Seckler, Bastian Steinert, Robert Hirschfeld
70	978-3-86956-230-8	**HPI Future SOC Lab - Proceedings 2011**	Christoph Meinel, Andreas Polze, Gerhard Oswald, Rolf Strotmann, Ulrich Seibold, Doc D'Errico
69	978-3-86956-229-2	**Akzeptanz und Nutzerfreundlichkeit der AusweisApp: Eine qualitative Untersuchung**	Susanne Asheuer, Joy Belgassem, Wiete Eichorn, Rio Leipold, Lucas Licht, Christoph Meinel, Anne Schanz, Maxim Schnjakin
68	978-3-86956-225-4	**Fünfter Deutscher IPv6 Gipfel 2012**	Christoph Meinel, Harald Sack (Hrsg.)
67	978-3-86956-228-5	**Cache Conscious Column Organization in In-Memory Column Stores**	David Schalb, Jens Krüger, Hasso Plattner
66	978-3-86956-227-8	**Model-Driven Engineering of Adaptation Engines for Self-Adaptive Software**	Thomas Vogel, Holger Giese
65	978-3-86956-226-1	**Scalable Compatibility for Embedded Real-Time components via Language Progressive Timed Automata**	Stefan Neumann, Holger Giese

ISBN 978-3-86956-265-0
ISSN 1613-5652