Babelsberg/JS

A Browser-based Implementation of an Object Constraint Language

Tim Felgentreff, Alan Borning, Robert Hirschfeld, Jens Lincke
Yoshiki Ohshima, Bert Freudenberg, Robert Krahn

VPRI Technical Report TR-2014-001

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

European Conference on Object-oriented Programming (ECOOP) 2014
Uppsala, Sweden, July 28-August 1, 2014, Springer

Babelsberg/JS

A Browser-based Implementation of an Object Constraint Language

Tim Felgentreff[1], Alan Borning[2,3], Robert Hirschfeld[1], Jens Lincke[1], Yoshiki Ohshima[3], Bert Freudenberg[3], Robert Krahn[4]

[1] Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
[2] University of Washington, Seattle, WA, USA
[3] Viewpoints Research Institute, Los Angeles, CA, USA
[4] Communications Design Group, SAP Labs, San Francisco, CA, USA

Abstract. Constraints provide a useful technique for ensuring that desired properties hold in an application. As a result, they have been used in a wide range of applications, including graphical layout, simulation, scheduling, and problem-solving. We describe the design and implementation of an Object Constraint Programming language, an object-oriented language that cleanly integrates constraints with the underlying language in a way that respects encapsulation and standard object-oriented programming techniques, and that runs in browser-based applications. Prior work on Object Constraint Programming languages has relied on modifying the underlying Virtual Machine, but that is not an option for web-based applications, which have become increasingly prominent. In this paper, we present an approach to implementing Object Constraint Programming without Virtual Machine support, along with an implementation as a JavaScript extension. We demonstrate the resulting language, Babelsberg/JS, on a number of applications and provide performance measurements. Programs without constraints in Babelsberg/JS run at the same speed as pure JavaScript versions, while programs that do have constraints can still be run efficiently. Our design and implementation also incorporate incremental re-solving to support interaction, as well as a cooperating solvers architecture that allows multiple solvers to work together to solve more difficult problems.

Keywords: Constraints, Object Constraint Programming

1 Introduction

Constraints are relations among objects that should hold. This could be that all parts in an electrical circuit simulation obey the laws of physics, that the rows in a Sudoku include each digit from 0 to 9, or that a streamed video plays smoothly in the presence of changing CPU and network load. We also want to support interactive use of constraints, for example, continuously re-satisfying a set of layout constraints on screen widgets as they are moved with

the mouse. In addition, it is useful to extend the constraint formalism to allow soft constraints as well as required ones, where the system should try to satisfy the soft constraints if possible, but it is not an error if they cannot be satisfied. For example, we might have a soft constraint for video quality that we are willing to relax if necessary, given the current network load, or a desired ideal spacing between two widgets that again can be relaxed if need be. In the work reported here, we want to support constraints in a clean way in an object-oriented language running in a lightweight, web-based programming environment.

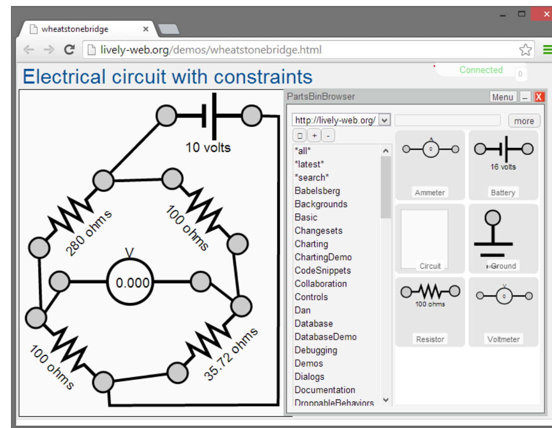

Fig. 1: Constructing a Constraint-based Wheatstone Bridge Simulation

Figures 1 and 2 are screenshots from our prototype system that illustrate the kinds of capabilities we want. Both are constructed in the Lively Kernel environment [1], an entirely web-based programming environment built on JavaScript.

Figure 1 shows a constraint-based simulation of Wheatstone Bridge being constructed. (A Wheatstone Bridge is used to measure an unknown electrical resistance by balancing two pairs of resistors so that the electrical potential between them is 0.) Parts representing batteries, resistors, and meters are copied from the Lively Kernel parts bin [2] on the right, dropped into the circuit on the left, and wired together. These parts carry constraints representing Ohm's Law, Kirchhoff's Current Law, and so forth. The system automatically solves the constraints when the parts are first connected, and re-solves them if the battery's supply voltage or a resistance is edited, updating the voltage displayed by the meter. (See Appendix A for the implementation.)

Figure 2 shows a color chooser from the parts bin that can be used to create a color palette for a website. Users can specify the desired average hue and luminance using the sliders, as well as change each color individually using a color chooser. The system automatically updates the colors and sliders according to user input — for example, the hue slider adjusts when the user changes the luminance slider. The system supports incremental re-solving, so that colors

change smoothly when dragging the sliders. Furthermore, the constraints on hue, luminance, and specific color selection have different priorities. The system is allowed to make changes to individually selected colors or cause changes to other colors to keep the average hue and luminance constant, whereas dragging the sliders forces the system to use the new value. The widget has additional constraints that luminance and hue of each individual color must be at least 80% of the average luminance and hue, and that the system cannot set red, green, and blue values outside the range 0–1.

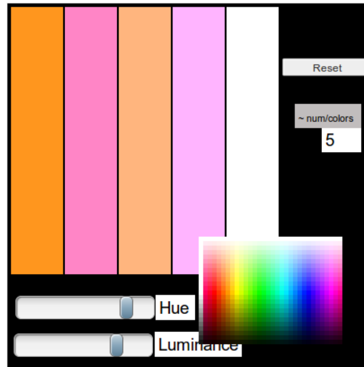

Fig. 2: Color Palette Chooser with Hue and Luminance Goals

While the capability to graphically construct constraint-based simulations dates back to Sketchpad [3] and ThingLab [4], in the current work we want to enable a true integration of constraints with the host object-oriented programming language, and further support this in a web-based environment. To accomplish this, we build on our recent work on Babelsberg [5], a language framework that supports an integration of constraint satisfaction with objects and their methods. Babelsberg in turn builds on earlier work on constraint-imperative programming in Kaleidoscope [6] and Turtle [7]. In Babelsberg, constraints are expressed as predicates using the underlying object-oriented language. The constraint is that the predicate evaluates to true, and the system maintains this constraint automatically whenever objects that participate in the constraint change.

Babelsberg improves on related approaches to constraint satisfaction in object-oriented programs, which use libraries [8,9,10], domain-specific languages [11,12], or (more recently) functional-reactive programming [13,14] to specify and solve constraints. These approaches do not need special runtime support, but require the programmer to call specific application programming interfaces (APIs) or follow certain rules to not accidentally circumvent the constraints.

We first implemented the Babelsberg design as a prototype in Ruby [15], called Babelsberg/R. This implementation depends on extending the Ruby Virtual Machine (VM). However, applications written in e.g. JavaScript typically have to work on a variety of client VMs included in different Web browsers. This

makes it infeasible to implement Babelsberg in a JavaScript VM. JavaScript is currently of considerable interest in the industry and research communities. Thus, an implementation as an extension written entirely in JavaScript enables us to apply constraint programming to a variety of existing problems, and to compare it directly with alternative solutions on a variety of platforms. Another goal for our design is good performance. As with the original Babelsberg/R implementation, we want the extension to have at most minimal impact on speed for programs without constraints; and for programs with constraints, we still want to have good performance for interactive graphical applications, which generally implies the need to support incremental constraint solvers [16].

In addition to the goal of good performance, a useful Object Constraint Programming language requires sufficiently powerful constraint solving capabilities. In prior work [5], we identified (but had not yet implemented) an important requirement, namely support for *cooperating constraint solvers*. The motivation is that it is often infeasible to provide a single constraint solver that works well for all aspects of a problem; instead, different solvers may be more appropriate than others for some aspects, and which need to work together to solve the problem. Our design and implementation in Babelsberg/JS provides this capability, in a way that supports incremental re-solving of constraints without requiring access to the VM.

The contributions of this work thus are:

- A design for Object Constraint Programming (OCP) languages that does not require VM support
- An implementation of cooperating constraint solvers, including techniques to do so without VM support and that support incremental constraint satisfaction
- A realization of these in an operational implementation in JavaScript, running in the Lively Kernel environment, including additional support within the language extension for writing constraint programs

The rest of this paper is structured as follows. Section 2 describes related work and the Babelsberg framework on which we build. In Section 3, we describe the features a language must provide to support Babelsberg without VM extensions. This design is realized in Babelsberg/JS (Section 3), which also includes support for cooperating constraint solvers (Section 3.1) and incremental re-solving (Section 3.2). We then describe the implementation of Babelsberg/JS in the Lively Kernel environment (Section 4), and the results of performance evaluation (Section 5). Section 6 describes future work and concludes.

2 Background and Related Work

Programs frequently have some set of constraints that should hold. In a standard imperative language, 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 may be implicit in the code and just expressed explicitly in comments and documentation, or perhaps in the form of machine-checkable assertions.

For some constraints, programmers may write assertions to fail early if the constraints are unexpectedly not satisfied [17], while other constraints describe invalid system states that can be automatically corrected. In our color chooser example, when the user selects a specific color that does not meet the luminance constraints, the system is allowed to change it. To deal with these kinds of situations, programmers may write corrective code that is executed at various times (for example, color adjustments may run while the user is dragging the luminance slider). This code uses branches and state changing operations to check and correct invalid state. However, these statements are order dependent, and the branching code expresses the constraints implicitly. Furthermore, it can be unclear whether a solution is complete in that it covers all possible cases or optimal. As argued in our prior work on Babelsberg [5] and elsewhere, it is usually clearer to express and satisfy constraints explicitly, rather than encoding them in control flow.

One approach to making the constraints explicit is to use a library that provides one or more constraint solvers. Numerous solvers, covering a wide range of type domains (including reals, booleans, and finite sets), are available for imperative languages and can be called from imperative code [8,9,10,16]. For more specialized domains such as user interface layout, some libraries provide separate domain specific languages (DSLs) to express, for example, minimal distances between graphical objects. Prominent examples here include the Mac OS X layout specification language [11] or the SQUANDER framework [18]. For our example, these approaches replace the branching and state changing code with declarative constraints. However, these constraints are expressed in the language of the library, using solver-specific types and expressions. To interact with the imperative state of the system, the solver must be called explicitly, and the constrained values must be copied between the solver and runtime data structures whenever either imperative code or the solver update them. This is error-prone, because programmers may accidentally circumvent the solvers if they do not call the solver in all required places. Further, because solvers often operate on a limited number of domain-specific primitive types, object-oriented abstractions cannot be used to express constraints.

An alternative approach is to integrate a means to express and maintain one-way constraints with the language itself. Some languages such as Scratch [19], LivelyKernel [1], and KScript [13] have built-in support for data flow, which allows programmers to express unidirectional constraints among objects. Babelsberg/JS shares with these systems the need to intercept object access and solve constraints when the system is disturbed.

To support a broader set of constraints, other languages directly integrate one or more solvers into their execution model. Again, there is a large body of prior art in this area, including Constraint Logic Programming [20], Constraint Imperative Programming [6,7], and Object Constraint Programming [5].

As illustrated by the color palette chooser example, it can be useful to extend the concept of constraints to include soft constraints with different priorities as well as required ones. There are various ways to formalize multiple priorities for soft constraints, and how to trade off conflicting soft constraints with the same priority; here we use the formalism described in [21]. In addition to hard and soft constraints, it is useful to add support for placing a *read-only annotation* on a variable in a given constraint. Operationally, a read-only annotation tells the system that it may not change the value of that variable to satisfy the given constraint.[1] Another useful extension is the addition of *stay constraints* and *edit constraints* [8], which provide important tools for integrating constraints with a language with state and in supporting interactive constraint systems. Stay constraints specify that a variable should keep its previous value. Soft stay constraints with a very low priority are used to express frame axioms, i.e., the desire that things remain the same unless there are some other constraints that force them to change. For example, suppose we are moving one part of a geometric figure with constraints. Without weak stay constraints to try and keep things where there used to be, the entire figure might collapse to a single point (still satisfying all its required constraints, but to the surprise of the user). Finally, edit constraints provide a concise and efficient way to support incremental updates, for example, moving a constrained object with the mouse. A typical sequence of actions when moving a part of a constrained figure is to first add edit constraints on the x and y values of a point being moved, then repeatedly provide new x and y values given the mouse position (and let these values propagate through the other constraints), and then finally remove the edit constraint when done moving.

2.1 Object Constraint Programming and Babelsberg

Object Constraint Programming differs from Constraint Imperative Programming in that it focuses on object-orientation as the main paradigm. It seeks to integrate declarative constraints in a way that does not compromise the expectations of imperative object-oriented programmers and that provides a declarative semantics that is compatible with these expectations.

Babelsberg is a design for a family of Object Constraint Programming languages. Since the language we present in this paper is an instance of this design, we summarize its goals in this subsection. These include:

- a syntax and semantics that are a strict superset of and fully compatible with the base language
- a unified mechanism for abstraction shared between constraints and object oriented code, so that constraints can re-use object-oriented methods and respect encapsulation

[1] For example, if we have a constraint $a + b = c?$, where c has been annotated as read-only, the system may change a or b or both to satisfy the addition constraint, but not c. Other constraints might change c, however, which would of course force changes to a or b. For simplicity, here we have given an intuitive, operational description of read-only annotations; please see [21] for a formal, declarative semantics.

- performance that is competitive with the base language for standard object-oriented code without constraints
- support for both required and soft constraints, constraints on object identity, variables that are read-only to solvers or imperative code, as well as incremental re-solving for use in interactive applications
- an API for constraint solvers that makes it straightforward to add new solvers and does not privilege the solvers provided with the implementation, to make it easy to use different solvers in different programs

2.2 Babelsberg/R

In [5] we describe Babelsberg/R, an implementation of Babelsberg based on a modified Ruby VM. The modifications are almost all semantic extensions, with only one minor syntactic extension, plus libraries for constraint satisfaction. 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, blocks (closures), object identity, and the language's control structures.

All these Ruby constructs are also supported in constraints. However, there are two important restrictions:

- The expression that defines a constraint should return a boolean, just like an assertion. The constraint is that the boolean is *true*.
- Constraints can be placed on the results of message sends, as long as the execution of these messages does not have side-effects (or those side-effects are benign, like caching), and repeated execution of the expression produces the same result, as long as no variables participating in the constraint have changed (so system calls for example to a random number generator or a file stream do not qualify)

For example, the constraint in the color chooser that each color should have a luminance at least 80% of the global targeted average can be expressed concisely in Babelsberg/R:

```
1 colors.each do |color|
2   always { palette.target_luminance * 0.8 <= color.luminance() }
3 end
```

What looks like an assertion on each element in the `colors` collection is actually a constraint. Whenever any color or the target palette luminance changes, the system will automatically adapt to ensure that this constraint is always satisfied. This snippet also shows that constraints can be used within imperative constructs and constrain the values of properties (the `target_luminance`) as well as the results of object-oriented message sends (the result of the calculated `luminance` of colors).

Given a set of constraint expressions, Babelsberg can choose among multiple solvers to find a solution to them. The architecture makes it straightforward to add new solvers, and does not privilege the solvers provided with the language (they are merely the ones that come with the standard library). However, the

programmer has to indicate which solver is available to the runtime, and there may be constraints that are too difficult for the solvers. Additionally, features such as incremental solving, read-only variables, soft constraints, and stay constraints are only available with some solvers.

Babelsberg/R was implemented by modifying the Ruby VM. It uses two interpretation modes: imperative evaluation mode and constraint construction mode. The interpreter normally operates in imperative evaluation mode. In the absence of constraints, this is the standard Ruby VM. However, if the interpreter encounters a LOAD or STORE instruction for a variable with a constraint on it, rather than directly loading or storing into the variable, it calls the appropriate constraint solver to retrieve the variable's value or to solve an equality constraint between the variable and the new value. When a constraint is being added, the interpreter switches to *constraint construction* mode. It continues to evaluate expressions using message sends, but rather than computing the result, it builds up a network of primitive constraints that represent the constraint being added, keeping track of the dependencies in the process.

To support this, the Ruby VM was extended to support *constrained variables*. These variables refer to different objects depending on the context they are used in. One is the normal object-oriented binding used in the host language execution. The other is a constraint object that can be used by a solver for constraint construction and solving. Variables become constrained variables only when they are used in a constraint, minimizing the performance impact for parts of the program where only normal variables are accessed.

3 Object Constraint Programming Without VM Support

In industry, JavaScript has become the de-facto standard for Web programming, and a huge amount of code exists in the language. This fact, along with JavaScript's unique design and its execution environment in a Web browser, also make it of great interest to the research community, motivating work on revising and adapting useful features of other languages to include in it [22,23].

To provide practical support for OCP in JavaScript, we adapt the Babelsberg design to not require support from the underlying VM. This enables us to run Babelsberg/JS in modern browsers and use it in a variety of practical Web applications.

For Babelsberg/JS, since we do not have access to the VM, we cannot redefine the operation of LOAD and STORE instructions to handle variables with constraints on them. Instead, the unmodified JavaScript VM is used only for imperative evaluation mode. To intercept accesses and assignments to constrained variables, we wrap properties with property accessors that interact correctly with the constraint solver. To get the value of a constrained variable, the accessor gets the value for that variable from its solver. For a store, the setter in general calls the appropriate constraint solver to solve an equality constraint between the variable and its new value for a store.

For constraint construction mode, we use a custom JavaScript interpreter, itself written in JavaScript. This custom interpreter is about three orders of magnitude slower than the underlying one. However, since evaluating code in constraint construction mode is a much less common activity, and one that doesn't occur in inner loops, the performance penalty is not a significant issue.

Generalizing our approach, we have thus identified the following requirements for implementing the Babelsberg scheme without VM support:

- The host language must support a means to intercept variable lookup, so names can refer to different objects.
- The VM-based implementation of Babelsberg assumes that the VM provides access to the program state so solvers can ignore encapsulation and modify data structures directly. In contrast, here the extension must enable calling the appropriate API functions to manipulate data structures.
- The host language must provide a means to modify interpretation of a block of code to implement the constraint construction mode.

The first requirement is only partially supported in JavaScript, namely for object fields using property accessors. We therefore limit ourselves to constraining field storage in Babelsberg/JS, but not storage into local variables. (Some compiled OO languages, for example C#, also support property accessors; and other dynamic OO languages, such as Python and Smalltalk, support *method wrappers* to enable intercepting accessors, again within the limitation of only constraining field access.) As with the original Babelsberg/R design, it does not matter whether the fields are constrained directly or whether they are used in the execution of a method that was constrained to produce a certain result. A property that is accessed in the execution of a constraint expression is wrapped with property accessor that intercepts lookup and storage.

Property Accessors for Constrained Objects When an object has been used in a constraint, its constrained properties have been replaced with property accessors. The *property getter* is a simple wrapper that reads from the solver variable in the most upstream region in which the field is referenced (cf. 3.1). Instead of returning the field value of the object, it returns the value of that variable in the solver data structure. The *property setter* distinguishes two cases. If the variable is writable from a solver, an equality constraint for that solver is created and the updated constraint system is solved, potentially triggering other solvers. On the other hand, if the variable is not writable (either because it is of a type that no available solver supports or because it has been marked as read-only by the programmer), its new value is stored, and all dependent constraints are recalculated. These dependent constraints have treated the variable as a constant (because they cannot modify it). To recalculate them, the constraints are deactivated in the solvers, and the expressions that created them are re-evaluated in constraint construction mode to create new constraints based on the new value. (The implementation of edit constraints (Section 3.2) handles the situation of repeated changes much more efficiently.)

Creating Constraints As an example of defining constraints, consider an interactive temperature converter, which maintains the relation between sliders representing values on the Fahrenheit, Celsius, Rankine, and Kelvin scales.

```
1 var converter = {},
2     cassowary = new CLSimplexSolver();
3 always: { solver: cassowary
4     converter.C * 1.8 == converter.F - 32 &&
5     converter.C + 273.15 == converter.K &&
6     converter.F + 459.67 == converter.R
7 }
```

In Babelsberg/JS, a source-to-source transformation creates a call to a global function — `always` — from an `always:` expression of this form (this transformation just provides syntactic sugar – the function can also be called directly with function object.) Once this function has executed, a change to any one of the temperature values in the converter object will trigger changes to the other three values to keep the constraint satisfied through property accessors described above.

The `always` function passes the predicate expressing the constraint and information about the context into a custom JavaScript interpreter. This interpreter is used to evaluate expressions in constraint construction mode, which is provided as part of the Babelsberg/JS library. The custom interpreter creates property accessors (getters and setters) for the `C`, `F`, `K`, and `R` fields of the `converter` object. The appropriate accessor is then called whenever some other part of the program uses one of those fields. However, within the constraint expression, accesses to these fields do not use these accessors, but instead return *ConstrainedVariable* objects. Messages are then sent to these objects, and instead of calculating values, build up networks of primitive constraints that can then be satisfied by a solver. The `always` function returns a `Constraint` object that provides meta-level access to the asserted relations, using the protocol described for Babelsberg/R [5].

In this example, the constraints are on the fields of the object. However, constraints in Babelsberg/JS (as with any instance of the Babelsberg scheme) can also invoke methods that perform computations. For example, imagine the converter uses the `getCelsius` method to return a cached temperature value that is updated in regular intervals from a Web service:

```
1 var converter = {},
2     cassowary = new CLSimplexSolver();
3
4 converter.getCelsius = function() {
5     if (!converter.updater) {
6         updateCelsius(converter); // updateCelsius omitted for brevity
7         converter.updater = setInterval(5000, function() {
8             updateCelsius(converter);
9         });
10    }
11    return converter.C;
12 }
13
14 always: { solver: cassowary
15     converter.getCelsius() * 1.8 == converter.F - 32 &&
16     converter.getCelsius() + 273.15 == converter.K &&
17     converter.F + 459.67 == converter.R
18 }
```

By placing the constraint on the result of sending messages rather than on fields, Babelsberg respects object encapsulation. The value returned from the message send in this example is simply a float, but return values can also be arbitrary objects and computed values. For example, we could constrain the maximum pressure of a volume of dry air with a fixed density and gas constant, which would effectively limit the maximum temperature to around 36° Celsius.

```
1 converter.pressure = function () {
2     var gasConstantDryAir = 287.058, // J/(kg * K)
3         density = 1.293; // kg/m^3
4     return density * gasConstantDryAir * converter.K / 1000;
5 }
6
7 always: { solver: cassowary
8     converter.pressure() <= 115 // kPa
9 }
```

3.1 Cooperating Constraint Solvers

The temperature converter described above has no graphical representation. Cassowary only works on reals, yet in order to display the temperature scales, we need to convert the values into strings and update the Web browser's Document Object Model (DOM) using the appropriate API. This is best done with a local propagation solver, which can invoke arbitrary methods to satisfy the constraints, in this case by calling the API. (The constraints that define the temperature converter are simple enough that we could have used a local propagation solver for all of them, but this is unsatisfactory for many problems, such as the Wheatstone bridge example in Figure 1, since local propagation cannot handle such situations as simultaneous equations or inequalities.)

There is currently no single solver that can efficiently handle all constraints that arise in a typical application (and it seems unlikely that one can be created). To address this, we extend the work presented in [5] to include an architecture for cooperating constraint solvers, allowing a problem to be partitioned among multiple solvers. For this example, we use two solvers: one for linear arithmetic on the reals, and one for local propagation constraints.

Our architecture for cooperating solvers partitions constraints into regions that are connected via read-only variables, implementing the design proposed in [24]. The result is a very loose coupling among the cooperating solvers. This approach is in contrast to the more commonly-used Satisfiability Modulo Theory (SMT) technique for supporting cooperating constraint solvers [25], which uses inferred equality constraints as the means for the cooperating solvers to communicate (including the case when neither of the equated variables has a specific value). Our experience so far indicates that our approach is more suited to integration with imperative constructs, in which variables do always have specific values, and lends itself well to support edit constraints for incremental re-solving. (While we have not yet done so in our implementation, the architecture described in [24] in fact allows hierarchies of cooperating solvers, so that within a single region, there could be multiple solvers that cooperate by sharing inferred equality constraints.)

In the cooperating solvers architecture, each constraint belongs to exactly one solver. All constraints that belong to the same solver are in the same region. While constraints belong to exactly one region, variables may be shared across regions. This happens if variables occur in multiple constraints that belong to different regions. These variables must be read-only in all but one of the regions. Read-only variables are represented in a solver-specific manner, either using stay constraints for solvers that support them, or through required equality constraints. To support this, solver libraries should provide a method that makes a variable read-only for them.

In this architecture, the regions must form an acyclic graph, so that solving can simply proceed from the upstream to the downstream regions, propagating variable values. Figure 3 shows an example configuration. Solving proceeds from the left and each solver propagates values for its variables to downstream solvers that need them. The downstream solvers can only read, not write to those variables. This architecture prohibits loops and a system that oscillates without finding a solution. To create this graph, the system determines an order for the solvers based on the dependencies between the constraints. The programmer can explicitly control the position of a solver in this graph, or the libraries can provide information so the system can create the order without the programmer's support. Applications can use multiple instances of the same solver type that are used one after the other (for example, for a problem that first uses Cassowary to solve simultaneous linear constraints, then DeltaBlue for local propagation constraints, then Cassowary again).

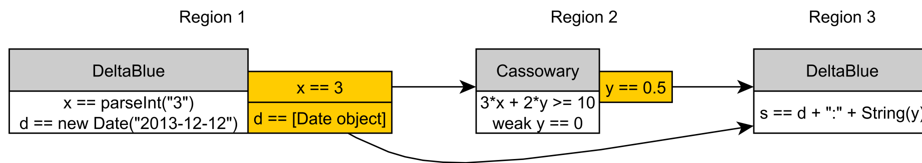

Fig. 3: Regions propagating variable values downstream

Once the solver regions are sorted, solving proceeds from the furthest upstream region. Each region will determine values for the variables it can write to, and the downstream regions will adjust to accommodate the new values propagated to their read-only variables from higher level regions. Soft constraints are solved for just within each region — in keeping with the theory of hard and soft constraints in the presence of read-only variables [21], if a soft constraint in an upstream region restricts a variable to a certain value, then a downstream region must use that value and can in fact not distinguish if this value was determined by a required or a soft constraint. If constraints in a downstream region cannot be satisfied due to an upstream soft constraint, we do not backtrack.

Given these additional capabilities, we can now add a graphical representation to our temperature converter. We want the color of a DIV element to change when the temperature is above 30° C.

```
1 var el = jQuery("#tooHotWarning");
2
3 always: { solver: deltablue
4     el.color.formula([converter.getCelsius()], function(celsius) {
5         var color = celsius > 30 ? "red" : "blue";
6         el.setAttribute("class", color);
7         return color;
8     });
9 }
```

Note that for the DeltaBlue local propagation solver, we do not provide a predicate (although we could — in that case it would be run to test whether re-solving is necessary). Instead, local propagation solvers need formulas for all writable variables that state their dependencies and how to update the variable. In this case, we want the Celsius value to be used as input for the color, but not vice versa, so we only provide one formula. The only dependency here is on the return value of `converter.getCelsius()`, passed explicitly in line 4. (Note that this could be omitted Babelsberg/R, because its version of DeltaBlue supports deducing the dependencies from the formula function — a feature we have not yet implemented here.) The dependencies are passed as arguments to the formula function, so we can use them directly to update the DOM using the browser's `setAttribute` API and return the new value. These functions, just like the predicates for Cassowary, are evaluated in constraint construction mode which wraps variables with property accessors — the function `formula` is simply a function defined by DeltaBlue.

3.2 Incremental Re-Solving for Cooperating Constraint Solvers

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, in which 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, including DeltaBlue and Cassowary, support this using edit constraints that allow a new value for a variable to be repeatedly input to the solver.

The original Babelsberg design did not include support for incremental resolving at the language level — it was up to the solver library to provide access to such functionality. However, to integrate with our cooperating solvers architecture, Babelsberg/JS does include support for incremental re-solving through a solver-independent `edit` function that takes the variables to be edited and returns a callback function. The process that produces new values can use this callback to input new values into the solvers for the variables to be edited.

The `edit` function gathers all the constraints in which the passed variables participate. Only variables that occur solely in solver regions that support edit constraints can be edited; otherwise an exception is raised. The read-only annotations for variables in the solvers for downstream regions are converted to edit

constraints, reflecting the fact that the upstream regions will be providing new values for these variables. Finally, the edit function creates a callback function and returns it. This callback can then be used to feed new values into the solvers.

As an example, suppose we wanted to connect the Celsius value of our temperature converter to a graphical slider. We wrap the original onDrag (which updates the slider's value) to input the new value into the edit callback as well.

```
1 var callback = edit(converter, ['C']);
2 slider.onDrag = slider.onDrag.wrap(function (originalOnDrag, evt) {
3     originalOnDrag(evt);
4     callback([slider.value]);
5 });
```

Two restrictions apply to the use of incremental re-solving with cooperating solvers: first, all variables that are edited must be only in regions of solvers that support edit constraints; and second, while the edit callback is used, no new constraints can be created. (Edit constraints are just a technique for optimizing the sequence of repeatedly replacing a constraint that a variable equal a constant with a new constraint with a new constant. Thus, if the restrictions aren't met, it is still possible to express and solve the desired constraints, just not as efficiently.)

4 Implementation in Lively Kernel

We have implemented Babelsberg/JS in the Lively Kernel environment [1]. We provide pure JavaScript implementations of DeltaBlue and Cassowary as constraint solvers and extend the Lively Kernel JavaScript interpreter to evaluate constraint expressions. The code is not Lively specific – we use the collection APIs and class system of Lively, but this could be trivially changed. However, when used in the Lively environment, we provide a source transformation that makes writing constraints in the Object Explorer [2] more convenient.

4.1 Assignment

Assignment to objects that are constrained in Babelsberg/JS is the core concept that binds the declarative constraints and imperative code together. Whereas in standard imperative code an assignment writes a value to a memory location, assignments in Babelsberg add equality constraints on constrained objects and trigger re-satisfaction. The new equality constraint may be unsatisfiable, in which case the assignment is not executed and a runtime exception is generated.

As in Babelsberg/R, the Babelsberg/JS runtime informs the developer of a failed assignment by generating a runtime exception. To support the cooperating solvers design, assignment in Babelsberg/JS is a 3-step process:

Set Value If the new value is the same as the old, we simply return. Otherwise, we convert all read-only constraints on the assigned variable either to required edit constraints (for solvers that support them) or to equality constraints.

If the assigned value has an external variable, i.e., it is constrained by a solver that can handle its type (for example a real in Cassowary), the new value

is input into the furthest upstream solver using an equality constraint and this solver is then called. Afterwards, the equality constraint for assignment is removed. However, if the solver cannot satisfy its constraints with the new value, an exception is raised.

Update Downstream Variables For all external variables except the primary one (the one in the most upstream solver), the new value is input into the solver. If any of the solvers fail to satisfy their constraints with the new value, an exception is generated and all read-only constraints are re-enabled as above.

All remaining constraints are in solvers that cannot handle the type of that variable. Consequently, its value was treated as a constant in their constraint expressions. With the new value, these have to be recalculated. The old variable value is remembered and the new value is stored. These constraints are disabled, their expressions re-evaluated in constraint construction mode, and then the constraints are re-enabled. If any constraint in this set fails to run its expression or cannot be satisfied with the new value, the old value is restored and an exception is generated.

Update Connected Variables Finally, we have to update the variables connected to the assigned variable. To do so, we create the transitive closure of all variables connected to the assignee through constraints. For all these variables, a new value has already been created for all solvers that already ran, but their downstream read-only constraints still have to be updated. These variables have to go through the first two steps of the assignment process, returning early if their values have not changed.

At this point, assignment can only fail for variables that are in solvers that have not run yet. These are only solvers that the primary assignee is not part of. If any one of these solvers cannot satisfy their constraints with the new value, we restore the old value, re-satisfy the constraints, and raise an exception. While this may leave the system in a different state than it was in before assignment (depending on the implementation of the participating solvers, they may not deterministically find the same solution to the same set of constraints) the system will still be in a state that satisfies all previous constraints.

Deferred Assignment of Connected Variables Babelsberg/R included an optimization to defer copying the values from a solver to the object-oriented variable location after assignment. Instead of copying the values for all affected variables immediately, the variable's values would be copied when they are next used in imperative code. This optimization cannot be used with our cooperating constraint solver architecture. Consider the following contrived example:

```
1  var obj = {a: 10, b: 10, c: true};
2  always: { solver: cassowary
3      obj.a + obj.b==20
4  }
5  always: { solver: deltablue
6    obj.c.formula([obj.b], function (b) {
7      if (b==13) throw "unlucky";
8      return b < 10;
9    })
10 }
11 obj.a = 7;
```

When `obj.a` changes, Cassowary is called to resatisfy the first constraint. However, to trigger DeltaBlue to solve the second constraint, the new value for `obj.b` has to be copied immediately, rather than when `obj.b` is next read. Otherwise, a failure to satisfy the second constraint is only encountered sometime later in the execution and difficult to trace back to the assignment that caused it.

Assigning Mutable Objects So far we have described how assignment is handled for atomic objects, such as integers and floats. We have not, in our design, addressed the case of mutable objects with substructure. Consider the following midpoint line:

```
1  var line = {start: pt(0,0), end: pt(1,1), midpoint = pt(0,0)};
2  always: { solver: cassowary
3    var center = line.start.getPosition().
4                    addPt(line.end.getPosition()).scaleBy(0.5);
5    line.midpoint.getPosition().eqPt(center);
6  }
7
8  line.midpoint = pt(1, 1);
```

There are two ways to look at such an assignment: a) the assignment asserts equality between `midpoint` and `pt(1, 1)` — both mutable objects — not their `x` and `y` parts. So the solver could also modify the parts of the newly assigned point to satisfy the constraint. This seems counter-intuitive, so presumably the right-hand side of an assignment should be read-only to the solver. On the other hand, there are use-cases for constraints for example in input rectification [26], where programmers may expect the system to fix the assigned object, rather than reject the assignment.

For now, we consider the behavior in this case to be implementation defined. Babelsberg/JS marks the assigned objects' parts with strong stay constraints. This means that, as long as other constraints allow, the solver will not change the new position for the midpoint. The design of a general solution is subject of further research.

Changing the Type of Variables Most solvers only provide support for a limited number of type domains (such as reals or booleans). When variables are used in constraints, their current values determine how they are handled by the solvers. Changing the type of a variable, although possible in a dynamic language, is a relatively uncommon operation, so slow performance is acceptable. When it does occur, the variable is removed from all solvers, all its constraints are

disabled, and its constraint expressions are re-executed in constraint construction mode, thus creating new solver-specific representations.

4.2 Constraint Construction

When a programmer writes a function that contains a constraint expression, this expression is evaluated using our JavaScript *ConstraintInterpreter*. Popular JavaScript VMs[2] (Apple Safari's SquirrelFish[3], Google Chrome's V8[4], Mozilla Firefox's SpiderMonkey [27], or Microsoft's Chakra[5]) do not provide direct access to the native interpreter or execution context of the caller, so our interpreter cannot look up names used in the constraint expression in the caller's environment. Instead, those names have to be passed explicitly.

Babelsberg/JS provides a source-to-source transformation based on *UglifyJS*[6], which collects names from the context and modifies the source code to pass those names into the constraint expression. This source transformation is enabled automatically when programmers use Babelsberg/JS in the Lively Kernel's Object Editor. For other JavaScript code, they have to provide a context object explicitly.[7]

Given a context, a function, and a solver, the ConstraintInterpreter executes the expressions in the function to create the constraint. The ConstraintInterpreter subclasses a JavaScript interpreter, modifying its behavior in three main aspects:

1. Slot accesses are intercepted. For each slot accessed during the execution of a constraint expression, property accessors are created that delegate access to a *ConstrainedVariable* object. For each slot, only one ConstrainedVariable is created on first access. ConstrainedVariables manage the communication with the various solvers and create solver specific representations of the slot value.
2. Certain unary (`!` and `-`) and binary operations (arithmetic, equality, inequalities, conjunction) are not interpreted as usual if an operand is a ConstrainedVariable or an expression involving ConstrainedVariables. Instead, the constraint object is sent a message to construct a solver-specific expression representing the operation and that expression is returned. For example,

---

[2] http://www.w3schools.com/browsers/browsers_stats.asp
[3] http://trac.webkit.org/wiki/SquirrelFish
[4] https://code.google.com/p/v8/
[5] http://en.wikipedia.org/wiki/Chakra (JScript engine)
[6] http://lisperator.net/uglifyjs/
[7] Note that we cannot use `eval` to access the outer scope. If we only supported constraints that access fields of objects in the scope and do not call user defined functions, we could have rewritten the code and evaluated the constraint expression using JavaScript's `eval` function, which has access to the enclosing scope. Using a custom interpreter, however, allows us to easily instrument the execution of most user-defined functions, so we can use normal object-oriented methods in constraint expressions.

in Cassowary, the expression `a.value <= b.value` would return a *LinearInequality* object.
3. Functions invoked in the expression are also interpreted in the ConstraintInterpreter by default. However, the plain JavaScript interpreter is used if the receiver is a ConstrainedVariable. In that case, the call is executed using normal JavaScript execution semantics. This is required to avoid creating constraints on the state of the solvers themselves.

The responsibility of ConstrainedVariables during constraint construction is to pass calls to the appropriate solver. To that end, a ConstrainedVariable lazily builds a mapping from solvers to solver-specific representations of its value. During construction, if the programmer has explicitly selected a solver, this solver is asked to provide a value representation by sending the message `constraintVariableFor` with the value as argument. If no solver was provided, the value is sent the `constraintSolver` message. Solver libraries may override this message for types that they can operate on. If the value responds with a solver instance, this solver becomes the active solver for the currently constructed constraint and is asked to provide a representation, again by sending `constraintVariableFor`.

Whenever a new representation is created in this manner, the solvers are sorted to determine which region the variable belongs to. Only the solver responsible for this region may write to the variable; as far as all other solvers are concerned it is read-only.

4.3 Determining Cooperating Solver Regions

The architecture for cooperating constraint solvers requires that each variable must be read-only in all but one of the regions that it occurs in. Furthermore, the regions and associated solvers must form an acyclic graph.

In Babelsberg/JS, when a variable appears in a new solver, we gather the solvers for the variable and sort them into regions. The region information is stored as a property of a solver instance. This allows, for example, the use of multiple instances of the same solver in different regions.

The variable is marked read-only for all solvers except the one in the furthest upstream region, the *defining solver*. This means that new values are assigned by calling `suggestValue` on the defining solver, and that all other solvers are triggered (in descending order of regions) once the defining solver has resatisfied its constraints, as described in Section 4.1.

4.4 Edit Constraints

Since the original Babelsberg design did not include language-level support for edit constraints, these were supplied by the solver libraries. In Babelsberg/R, the meta-level protocol for inspecting constraints was used to support edit constraints in Cassowary and DeltaBlue. The programmer called the appropriate edit method with the objects to be edited and a stream that would provide new

values. The Constraint meta-protocol was used to create edit variables, constrain them to be equal to the supplied variables, and update them from the stream.

To support cooperating incremental re-solving (cf. Section 3.2), in Babelsberg/JS there are two changes to this scheme. First, to support edit constraints within a single thread, the edit method returns a callback to input new values into the solvers, rather than taking a stream of values. Second, since the language design now supports edit constraints explicitly, the solvers have to provide a specific edit constraint API.

Upon calling the edit method, the following methods are called on the solvers and the supplied variables, in order:

prepareEdit is called on each solver variable. In this method, variables can prepare themselves for editing. In Cassowary, for example, this would call the `addEditVar` method on the solver with the variable as argument. For DeltaBlue, this creates an `EditConstraint` on the variable and adds it to the list of constraints.

beginEdit is called once for each solver participating in the edit before the callback is returned. In Cassowary, this initializes the edit constants array and prepares the solver for fast re-solving when these constants change. In DeltaBlue, the solver generates an execution plan to solve the constraints starting with the EditConstraints as input.

Now the callback can be used to input new values into the system and trigger re-solving. The callback will call **resolveArray** on each solver with the new values and update the object's storage (so other observers and hooks around the values still work). Because the solver's execution plan is fixed for the duration of an edit, we disallow creating new edit callbacks before the current edit has finished. When new constraints are created, the execution plan may also become invalid, but we do not enforce invalidating the edit callback in this case.

To finish editing, the callback is simply called without supplying new values.

finishEdit is sent to each solver variable. Cassowary variables do nothing here, DeltaBlue variables remove their EditConstraints from the solver.

endEdit is called once for each solver to reset the solver state.

Compared to Babelsberg/R, this makes the interface for edit constraints uniform across solvers and also allows it to work with cooperating solvers. However, each solver now has to provide some support for this, so more work is required to enable the feature.

5 Performance Evaluation

Our design tries to provide reasonable performance for a variety of applications. To evaluate its performance, we investigated two scenarios: a) how constraint solving performance compares with using imperative code to satisfy the constraints, b) how object-oriented performance is affected by our extension (thus

comparing the use of Babelsberg/JS with calling a constraint satisfaction library from standard imperative code).

For the first problem, we used a Kaleidoscope example as a benchmark [28]. (The same benchmark was used for Babelsberg/R.) In this example, we simulate a user interaction in which the user drags a slider to adjust the upper end of the mercury in a thermometer. The constraints are that the mercury should follow the mouse if possible, but must not go outside the thermometer, and that the graphical representation of the thermometer and mercury (using a gray and a white rectangle) as well as a number displaying the current value, should be updated.

We compare the performance of a purely imperative solution using branches and assignments, a constraint version that calls the Cassowary constraint satisfaction library from imperative code, and a version with the same set of constraints in Babelsberg/JS (cf. Appendix B). Both of the constraint versions use edit constraints.

	Imperative	Library	Babelsberg/JS
100x	1.47 ± 0.128	24 ± 0.486	109 ± 2.29
1,000x	1.62 ± 0.0922	143 ± 4.26	214 ± 4.89
10,000x	1.86 ± 0.382	1445 ± 270	1311 ± 304

All numbers are the average execution time in milliseconds ± the standard deviation. We ran each set of iterations 10 times on Firefox 27 on a 3.2 Ghz Intel Core i5. This micro-benchmark show that, in extreme cases, the object-constraint versions are many hundred times slower than the purely imperative solution. However, Babelsberg/JS is comparable to the library-based approach. Using a library has less overhead for few iterations (where creating the constraints takes a large portion of the time in Babelsberg/JS). However, in both cases, by using edit constraints, we can achieve acceptable performance for repeatedly solving a set of constraints with varying input values. Considering that Babelsberg/JS is intended for imperative programmers who want to express constraints in some parts of their programs, 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. Furthermore, we think the benefits for comprehensibility, code size, and robustness justify the performance impact in some system parts. The imperative code is more complex because it has to make all cases explicit using branches, it is hard to tell whether the solution is optimal or complete, and the constraints are hard to derive from the code.

To test how the purely object-oriented parts of a system are affected if we pass objects with constraints to them, we measured the overhead of field access for constrained versus unconstrained fields by repeatedly reading the same 5 properties from an object first without and then with equality constraints on each variable. These results are comparable to those for Babelsberg/R and show

	Unconstrained Access	Constrained Access
1,000x	3.31 ± 0.289	8.57 ± 1.08
10,000x	20.4 ± 0.694	29.8 ± 1.82
100,000x	189 ± 5.83	241 ± 15.9

that the overhead for reading constrained objects in purely imperative parts of the code is minimal.

In our example applications — the circuit simulation, color palette chooser, and temperature converter presented above, as well as a simple particle simulation, an available-to-promise function, and a layout example — the overhead of constraints was much less pronounced and they provided interactive performance, often even without using edit constraints.

6 Future Work and Conclusion

We have presented a design for implementing an Object Constraint Programming language without VM support, which is realized as a JavaScript extension called Babelsberg/JS. We have also implemented a number of features from the original OCP design, including unified language constructs for constraint definition and object-oriented code, automatic maintenance of constraints, integration with the existing syntax and semantics, an interface to add new solvers and constraint solver constructs such as read-only variables and incremental re-solving; and also extended the design to support cooperating constraint solvers. There are a number of directions for future work.

Usability of Babelsberg/JS An important area for future work is the evaluation of the usability of our approach in general applications. We are interested in the comprehensibility of Babelsberg/JS code, especially to the target group for this language, i.e., imperative programmers with little prior experience with constraint programming. This will also provide opportunity to compare performance on more practical examples.

Debugging, Explanation, and Solver Selection It is currently difficult to tell why a solver may not be able to satisfy a given constraint, why it produced an unexpected result, or why finding a solution is slow. Our ConstraintInterpreter should include support for reasoning about the constraint system it builds. Prolog (or just a direct backtracking algorithm) may be useful as a "meta-solver" to automatically find a solver (or set of solvers) for a particular configuration of constraints.

Other Babelsberg/R features Babelsberg/R included support for more OCP features that we have omitted for now in this work. Specifically, we want to add support for identity [29], class, and message protocol constraints. Furthermore, to control when solving is invoked, Babelsberg/R provides multi-assignments to update multiple values simultaneously before a solver is invoked. Finally, we plan

to add the convenience methods `once` and `assert ... during ...` to control the duration of constraints, although these could be trivially added using the meta-protocol of `Constraint` objects.

Babelsberg/JS, compared to the earlier Babelsberg/R implementation, can be applied more directly to existing problems. It runs unmodified in different Web browsers, and integrates with the existing imperative language and libraries. The work reported here is quite recent, and we expect to continue to evolve both the language and its implementation.

References

1. Ingalls, D., Palacz, K., Uhler, S., Taivalsaari, A., Mikkonen, T.: The Lively Kernel – a self-supporting system on a web page. In: Proceedings of the Workshop on Self-Sustaining Systems (S3), Springer (May 2008) 31–50
2. Lincke, J., Krahn, R., Ingalls, D., Roder, M., Hirschfeld, R.: The Lively PartsBin– a cloud-based repository for collaborative development of active web content. In: 2012 45th Hawaii International Conference on System Science (HICSS'12), IEEE (January 2012) 693–701
3. Sutherland, I.: Sketchpad: A man-machine graphical communication system. In: Proceedings of the Spring Joint Computer Conference, IFIPS (1963) 329–346
4. Borning, A.: The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. ACM Transactions on Programming Languages and Systems **3**(4) (October 1981) 353–387
5. Felgentreff, T., Borning, A., Hirschfeld, R.: Babelsberg: Specifying and solving constraints on object behavior. Technical Report 81, Hasso Plattner Institute, Potsdam, Germany (May 2014) Also published as TR-2013-001, Viewpoints Research Institute, Los Angeles, CA.
6. Lopez, G., Freeman-Benson, B., Borning, A.: Kaleidoscope: A constraint imperative programming language. In: Constraint Programming. Volume 131. Springer-Verlag (1994) 313–329 NATO Advanced Science Institute Series, Series F: Computer and System Sciences.
7. Grabmüller, M., Hofstedt, P.: Turtle: A constraint imperative programming language. In: Research and Development in Intelligent Systems XX. Springer (2004) 185–198
8. Badros, G.J., Borning, A., Stuckey, P.J.: The Cassowary linear arithmetic constraint solving algorithm. ACM Transactions on Computer-Human Interaction (TOCHI) **8**(4) (2001) 267–306
9. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), Springer (March 2008) 337–340
10. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 4424. Springer (April 2007) 632–647
11. Sadun, E.: iOS Auto Layout Demystified. Addison-Wesley (October 2013)
12. Enthought Inc: Enaml 0.6.3 documentation (February 2014)
13. Ohshima, Y., Lunzer, A., Freudenberg, B., Kaehler, T.: KScript and KSWorld: A time-aware and mostly declarative language and interactive GUI framework. In: Proceedings of the 2013 ACM International Symposium on New Ideas, New

Paradigms, and Reflections on Programming & Software. Onward! '13, New York, ACM (2013) 117–134
14. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: A programming language for Ajax applications. ACM SIGPLAN Notices **44**(10) (2009) 1–20
15. Flanagan, D., Matsumoto, Y.: The Ruby Programming Language. O'Reilly (January 2008)
16. Freeman-Benson, B.N., Maloney, J., Borning, A.: An incremental constraint solver. Communications of the ACM **33**(1) (1990) 54–63
17. Rinard, M., Cadar, C., Nguyen, H.H.: Exploring the acceptability envelope. In: Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'05), ACM (October 2005) 21–30
18. Milicevic, A., Rayside, D., Yessenov, K., Jackson, D.: Unifying execution of imperative and declarative code. In: 33rd International Conference on Software Engineering (ICSE). (May 2011) 511–520
19. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al.: Scratch: programming for all. Communications of the ACM **52**(11) (2009) 60–67
20. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Proceedings of the 14th ACM Principles of Programming Languages Conference (POPL'87), ACM (January 1987) 111–119
21. Borning, A., Freeman-Benson, B., Wilson, M.: Constraint hierarchies. LISP and Symbolic Computation **5**(3) (1992) 223–270
22. Van Cutsem, T., Miller, M.S.: Proxies: Design principles for robust object-oriented intercession APIs. ACM Sigplan Notices **45**(12) (2010) 59–72
23. Kang, S., Ryu, S.: Formal specification of a JavaScript module system. In: Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications, ACM (2012) 621–638
24. Borning, A.: Architectures for cooperating constraint solvers. Technical Report VPRI Memo M-2012-003, Viewpoints Research Institute, Glendale, California (May 2012)
25. Nelson, G., Oppen, D.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems **1** (1979) 245–257
26. Long, F., Ganesh, V., Carbin, M., Sidiroglou, S., Rinard, M.: Automatic input rectification. In: 2012 34th International Conference on Software Engineering (ICSE), IEEE (2012) 80–90
27. Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., et al.: Trace-based just-in-time type specialization for dynamic languages. ACM Sigplan Notices **44**(6) (2009) 465–478
28. Lopez, G., Freeman-Benson, B., Borning, A.: Kaleidoscope: A constraint imperative programming language. In: Constraint Programming. Volume 131. Springer-Verlag (1994) 313–329 NATO Advanced Science Institute Series, Series F: Computer and System Sciences.
29. Lopez, G., Freeman-Benson, B., Borning, A.: Constraints and object identity. In: Proceedings of the 1994 European Conference on Object-Oriented Programming (ECOOP'94), Springer (July 1994) 260–279

A Examples

Circuits The circuit parts are represented by classes that create constraints in their initializers. (The context has to be passed because classes are written in plain JavaScript files in Lively without source transformation.) The code to connect leads is omitted (it constrains voltages to be equal and the sum of currents to be 0.0 between leads).

```
1  Object.subclass('TwoLeadedObject', {
2     initialize: function() {
3        this.lead1 = {voltage: 0.0, current: 0.0};
4        this.lead2 = {voltage: 0.0, current: 0.0};
5        always({solver: cassowary, ctx: {self: this}}, function () {
6           return self.lead1.current + self.lead2.current == 0.0;
7        });
8     },
9  });
10 TwoLeadedObject.subclass('Resistor', {
11    initialize: function($super, resistance) {
12       $super();
13       this.resistance = resistance;
14       always({solver: cassowary, ctx: {self: this}}, function () {
15          return self.lead2.voltage - self.lead1.voltage ==
16                 self.lead2.current * resistance
17       })
18     },
19 });
20 TwoLeadedObject.subclass('Battery', {
21    initialize: function($super, supplyVoltage) {
22       $super();
23       this.supplyVoltage = supplyVoltage;
24       always({solver: cassowary,
25              ctx: {self: this, supply: this.supplyVoltage}},
26          function () {
27            return self.lead2.voltage - self.lead1.voltage == supply
28       })
29    },
30 });
31 Object.subclass('Ground', {
32    initialize: function() {
33       this.lead = {voltage: 0.0, current: 0.0};
34       always({solver: cassowary, ctx: {self: this}}, function () {
35          return self.lead.voltage == 0.0 && self.lead.current == 0.0
36       })
37    },
38 });
39 TwoLeadedObject.subclass('Wire', {
40    initialize: function($super) {
41       $super();
42       always({solver: cassowary, ctx: {self: this}}, function () {
43          return self.lead1.voltage == self.lead2.voltage
44       })
45    },
46 });
47 TwoLeadedObject.subclass('Voltmeter', {
48    initialize: function($super) {
49       $super();
50       this.readingVoltage = 0.0;
51       always({solver: cassowary, ctx: {self: this}}, function () {
52          return self.lead1.current == 0.0 &&
53            self.lead2.voltage - self.lead1.voltage == self.readingVoltage
54       })
55    },
56 });
```

B Benchmarks

For comparing purely imperative to purely constraint-oriented performance we started with the following imperative version.

```
for (var i = 0; i < this.Iterations; i++) {
    mouse.location_y = i
    var old = mercury.top
    mercury.top = mouse.location_y
    if (mercury.top > thermometer.top)
        mercury.top = thermometer.top
    if (old < mercury.top) // move gray rect upwards (draws over the white)
        gray.top = mercury.top
    else // move white rect downwards (draws over the gray)
        white.bottom = mercury.top
    display.number = mercury.top
}
```

In the constraint library and Babelsberg/JS versions, we specify the same constraints and use an edit constraint in the same manner, once through the Cassowary API and once in the syntax of Babelsberg/JS. Given below is the Babelsberg/JS version. (The solver argument is omitted for brevity.)

```
always(function() { return display.number==mercury.top });
always(function() { return white.top==thermometer.top });
always(function() { return white.bottom==mercury.top });
always(function() { return gray.top==mercury.top });
always(function() { return gray.bottom==mercury.bottom });
always(function() { return mercury.top <= thermometer.top });
always(function() { return mercury.bottom==thermometer.bottom });
always({priority: "strong"}, function() {
    return mercury.top==mouse.location_y
});

var cb = edit(ctx.mouse, ["location_y"]);
for (var i = 0; i < this.Iterations; i++) {
    cb(i);
}
```

To compare accessor performance with and without constraints, we measured the following two loops individually:

```
var o = {get a() {return 0}, get b() {return 0}, get c() {return 0}},
    oc = {a: 0, b: 0, c: 0};
always({solver: cassowary, ctx: {oc: oc}}, function() {
    return oc.a==0 && oc.b==0 && oc.c==0
});

for (var i = 0; i < this.Iterations; i++) {
    sum = o.a + o.b + o.c;
}
for (var i = 0; i < this.Iterations; i++) {
    sum = oc.a + oc.b + oc.c;
}
```

C Artifact Description

Authors of the artifact. Design and documentation: Tim Felgentreff, Alan Borning, Robert Hirschfeld, Jens Lincke, Yoshiki Ohshima, Bert Freudenberg, Robert Krahn. Core developer: Tim Felgentreff.

Summary. The artifact shows Babelsberg/JS, an implementation of the Babelsberg design for object-constraint programming in the Lively Kernel. It includes an installation of the Lively Kernel environment and a number of example applications, some of which are mentioned in the paper. A screencast shows how the examples can be accessed. The provided package is designed to support repeatability of the experiments of the paper: in particular, it allows users to try and modify the example applications from the paper, as well as to run the benchmarks.

Babelsberg/JS uses a modified JavaScript interpreter to transform constraint expressions into constraints that are handed to the Cassowary and DeltaBlue constraint solver libraries. The full source code is included in the Lively Kernel environment, and instructions for exploring it are included.

Content. The artifact package includes:

- a Babelsberg/JS installation in a local Lively Kernel environment;
- the Chromium browser already open on a Lively Kernel world;
- a screencast that shows how to interact with the examples.

We provide a VirtualBox disk image for testing Babelsberg/JS. The image contains a stripped down installation of Ubuntu 13.10 LTS set up to launch Chromium directly with the screencast and the Lively Kernel page already open. Through port forwarding the environment is also accessible from the host: `http://localhost:9001/users/timfelgentreff/ecoop_artifact.html`. Note that to access the latter, we recommend a WebKit-based browser (Safari, Chrome, or their derivatives) or a recent version of Firefox (29 at the time of this writing).

Getting the artifact. The artifact endorsed by the Artifact Evaluation Committee is available free of charge as supplementary material of this paper on SpringerLink.

Tested platforms. The artifact is known to work on Oracle VirtualBox version 4 (`https://www.virtualbox.org/`) with at least 512 MB RAM.

License. BSD-3-Clause (`http://opensource.org/licenses/BSD-3-Clause`) for Babelsberg/JS, MIT (`http://opensource.org/licenses/MIT`) for the Lively Kernel environment

MD5 sum of the artifact. 57324cb58f7a517ab1abd1088bbd9d0f

Size of the artifact. 810 MB