Parallel Web Scripting with Reactive Constraints

Thibaud Hottelier
James Ide
Doug Kimelman
Ras Bodik

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-16
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-16.html

February 14, 2010

Copyright © 2010, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Parallel Web Scripting with Reactive Constraints

Thibaud Hottelier, James Ide,
Rastislav Bodik

University of California, Berkeley

{tbh,bodik}@cs.berkeley.edu,
ide@berkeley.edu

Doug Kimelman

IBM T.J. Watson Research Center

dnk@us.ibm.com

Abstract

We describe a preliminary design of a webpage layout and scripting language based on constraints and reactivity. The language is intended to play the role of CSS (layout) and AJAX (reactivity) in today's browsers, which together account for more than half of the CPU cycles, a performance bottleneck on smart phones. The rationale is to (1) integrate layout and reactivity in one language, thus eliminating the overhead of the frequent context switch between JavaScript and the layout engine; (2) allow web designers define custom layout semantics, freeing them from the inflexible CSS and allowing them to write layouts that adapt to a range of screen sizes; (3) parallelize layouts with the help of hierarchically expressed constraints; and (4) parallelize handling of multiple events by detecting that events have independent effects on the constraint system. We describe our semantics for combining events and constraint solving and illustrate, on a small case study, how we deal with under-constrained and over-constrained programs.

1. Motivation

In the 90's, high-level dynamically-typed, mostly-functional languages evolved from their scripting beginnings to full-fledged applications. This transition has been driven by Web programming, in particular JavaScript on the client, and Python CGI programs on the server. The popularization of these high-level languages was also enabled by abundant computing power, which in the 90s was doubling every 18 months, compensating for the performance inefficiencies of these languages, which in general were interpreted. Programmers have grown to appreciate the productivity benefits of high-level programming—for instance, the ease of embedding domain-specific languages such as jQuery into JavaScript and the powerful layout engine of the web browser—and we now see most new laptop and desktop applications written in the browser, with JavaScript frameworks, rather than as standalone desktop applications in C++. This is a good thing: high-level programming has raised the programming abstraction level and made it easier to continue doing so by simplifying the creation of domain-specific programming frameworks.

A potential obstacle to further proliferation of these productive languages and programming environments is the end of the exponential improvements in single-thread computing speed. In about 2002, the single-thread performance of laptop and desktop processors ceased to improve. The physics of CMOS transistors dictated that, from that point onwards, the benefits of shrinking transistor sizes have been realized as multiple cores. This trend is likely to continue even after CMOS is replaced with another technology: we'll see more parallelism and but the performance of a single thread is likely to remain fixed or be smaller, for the benefits of energy efficiency. Because applications written in high-level languages (such as Google Maps) are not easily parallelizable, single thread performance is what matters for them. The implication is that the situation of the golden era of the 90s has reversed: back then, we had more performance than high-level languages demanded; now the richness of applications written in high-level languages requires performance that we lack. As a result, Adobe, Google, Mozilla and WebKit have recently started working very actively on JIT compilers for their JavaScript virtual machines. A deeper analysis of how to support productive programming in the post-Moore's Law Era can be found in [3].

We are mostly concerned with high-level programming in the browser, which is now the dominant platform for application development on the laptop and the desktop, thanks to the convenient browser abstractions: the powerful layout engine controlled by JavaScript. The performance penalty of the abstraction is rather high (about 100x compared to a program written in C [3]). The result of this penalty is that programming on the phone could not afford to adopt the browser approach because the phone processors are performance lim-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $10.00

ited. Instead, programs for the iPhone and Android phones are written in lower-level languages and frameworks, which trade abstraction and expressiveness for lower overhead. Because we predict that a lot of our laptop activity will transition to phones (soon aided by laser pico-projectors), we are interested in providing a browser-like programming environments for these platforms—platforms whose performance will continue to lag behind that of laptops.

What should a language for productive application programming look like? The following are factors that led us to the design of the constraints-based language.

- The language must be amenable to *compilation* that can produce single-thread code with little abstraction overhead. In [3], we describe why compilation and partial evaluation of the current browser architecture is unlikely to be very effective in reducing the 100x runtime penalty. Our hope is that we can compile constraints into solver code that is free of artifacts present in code obtained from dynamically typed languages, such as frequent function calls and hashtable lookups.
- To support forward performance scalability as processors add more cores (even phone processors will be multicores), the language must allow easier *parallelization* than the dynamically typed JavaScript, which complicates discovery of program dependences. Our hope is that hierarchical constraints will allow us to identify constraints that we can solve in parallel.
- In today's browser, a lot of performance is lost in the frequent context switches between the document layout and scripting. Our goal, therefore, is to express in one program both the layout constraints and the reactive program logic that handles the user interface and the interaction with the server.
- Finally, we would like to raise the level of abstraction from the event-driven browser programming model to one that hides the explicit continuations of JavaScript callbacks. Constraints allow us to do that better than dataflow because they express bidirectional dependence in a single constraint.

2. Event-driven Constraint-Based Programming

This paper describes a language design that combines the constraint-based and event-driven programming models. When events are added to constraints, the language design choices seem to boil down to the question of how the constraint system evolves in response to a sequence of events. Specifically, if the constraint system represents the state of the program, how does an event modify the constraint system to reflect the state change?

To illustrate the interaction of constraints and events, consider the constraint system for converting Celsius temperatures to Fahrenheit temperatures and vice versa, adopted from [1] and shown in Figure 1. The constraints network in the figure establishes the familiar arithmetic constraints on variables represented by edges. To compute with this constraint system, the user interacts with it by setting values of variables and asking the solver to solve for the values of the remaining variables. Each such interaction can be thought of as the user sending an event to the system. For example, when C is set to 25, the system answers "$C = 25, F = 77$".

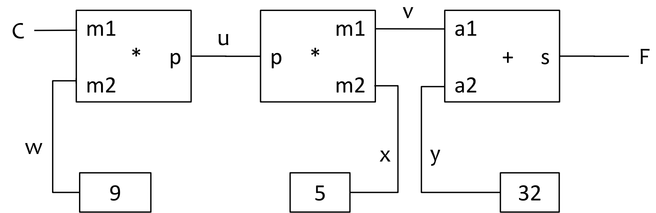

Figure 1. A constraints system for converting Celsius to Fahrenheit and vice versa [1].

The design choices become obvious when the user follows by setting the value of F to 212. What is a meaningful semantics? Should C be implicitly unset so that it becomes unconstrained and can be solved under the new user constraint $F = 212$? The semantics chosen in [1] keeps C set and the system hence must reject the user's attempt to set F because the values of C and F would be inconsistent with the constraints. Instead, to compute C from F, the user must first unset C, and then set F. Asking the user to explicitly unset a variable may sometimes be a suitable choice. In other scenarios, the system may want to react to an event by outputting a new constraints system [8].

In this paper, we investigate semantics suitable for programming with constraints and events in the web browser. Our design is driven by a case study with a prototypical Web 2.0 application that contains user interaction and combines multiple media (see Section 4).

3. The semantics

This section informally describes the semantics of our language. The semantics is illustrated in our case study in Section 5, where we show how constraints handle programming problems common in our domain and we also suggest additional language features. The design of our language is not complete; we are still investigating the implications of our design decisions and in Section 7 we discuss problems that still remain and how we plan to solve them.

We first describe the constraint system, then add events. For the sake of presentation, we will use the graphical syntax of constraint networks.

Constraints The constraint system consists of *variables* (also referred to as *ports*), *constraints* and *components* (see Figures 2). A constraint, represented by a rounded box with a list of ports, defines a relation that characterizes constraints imposed on the ports. Equality constraints can also be repre-

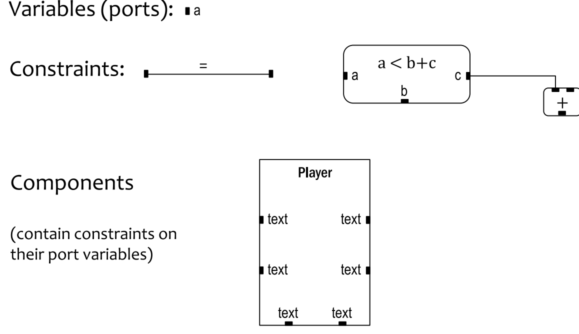

Figure 2. Building blocks of the constraint system.

sented by wires connecting the variables or ports of separate boxes. A component, represented as a squared box with a list of ports, encapsulates constrained networks, for example as shown in Figure 6. The list of ports are the public interface of the component.

A *solution* to the constraint system is a valuation of program variables that satisfies all constraints. At program start, some variables may be given initial values; the values of the remaining variables are solved to form a solution. Although the choice of the constraint language influences decidability, we are not assuming a particular language in this section. Our domain seems to require linear constraints with boolean connectives, and this is the language we use in our case study in Section 4.

Each variable has a static type. In our case study, we consider three basic types: integers, rationals, and booleans. Complex types can be constructed with lists and tuples as summarized below, using the ML type syntax.

$$\begin{array}{rcl} Type & ::= & Type\ \texttt{list} \mid Type * \ldots * Type \\ & & \mid Basic_Type \\ Basic_Type & ::= & \texttt{Int} \mid \texttt{Rational} \mid \texttt{Boolean} \end{array}$$

Structural equality is used to compare complex types. The type (or signature) of a component is the sum of the type of its ports.

The set of program variables can change at runtime through combinators such as *map* and *fold* that apply constraints, packaged as a component, onto a list of variables. The list can be of dynamic length, e.g., it can correspond to the changing content of a file or the value of an event. Lists are also used to define components with variable number of ports. For instance, the signature of VBox in Figure 9 is

$$(\mathrm{y} : \mathbb{Z},\ \mathrm{height} : \mathbb{Z},\ (\mathrm{y}_i : \mathbb{Z},\ \mathrm{height}_i : \mathbb{Z})\ \texttt{list})$$

The VBox component expresses the following constraints:

$$\begin{array}{ll} & y = y_0 \\ \wedge & \forall i \in [1, n].\ y_i = y_{i-1} + height_i \\ \wedge & height = y_n - y_0 \end{array}$$

where (y_n, h_n) represents the last element of the list. The network of constraints $y_i = y_{i-1} + height_i$ is generated using a *fold* combinator, which produces one such constraint per element of the list:

$$\mathrm{fold}(\lambda s.\ \lambda(y, h).\ \lambda res.\ s = y \wedge res = s + h,\ y, list)$$

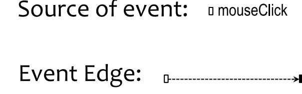

Figure 3. Events representation.

Events The reactive part of our language is modeled with events. In our domain, an event is the mechanism for the user interface or the server to request modifications to the web page, which is modeled by the constraint system. Semantically, an event is a request to change the value of a variable, which may in turn also change the structure of the constraint system using the combinators mentioned above. As shown by Figure 3, events are represented by dashed arrows between an event source and a variable. The source of an event is an external entity, e.g., a mouse. Like variables, sources of events have a type and an event edge may connect a source to a variable only if they have the same type.

An event is a pair (r, v) where r is a variable whose value is to be changed and v is the new value of r. When handling of an event has been completed, the constraint system is in a consistent state (i.e., variable values form a solution). The following five steps are taken to handle an event (r, v). These five steps happen atomically.

1. The variable r is asserted to have the value v. This assertion will in general be inconsistent with the current solution.
2. Compute R, the smallest set of variables whose values need to be changed to obtain a new solution from the previous solution.
3. Compute V, the valuation map that gives the new solution for the variables in R.
4. Set the variables in R according to V.
5. The assert on the variable v is removed. This does not change the solution because the system does not leave a particular consistent state without an event stimulus.

This event semantics minimizes the update to the state after an event. One can certainly imagine a different criterion for which variables to update (Step 2), and in fact we are not yet certain that this semantics is a good fit for all applications in our domain. We briefly discuss in Section 7 the alternatives for solving steps 2 and 3.

Properties. *Unique Solution.* We are in general interested in programs whose constraint systems always have a solution and never have multiple solutions. The system should have a solution both in the initial state and after every event. We discuss this issue in Section 5.

Hidden variables. A component typically contains variables, called *hidden variables*, that are not observable from outside the component. These variables may influence the constrains imposed on the ports of the component. For example, a hidden variable may select between two sets of constraints on the ports. Hidden variables may complicate program analysis, even in a system that has always has unique solutions. When a hidden variable encodes *state*, i.e., it is not a function of the port values, but instead a function of the sequence of values on the ports. Such hidden variables are the reason why components in general cannot be described as a relation on the port values. Instead, the history of values must be considered to model their behavior.

4. The Case Study

To give the reader a feel for the language, we present how one would write a small video player application using constraints. As shown in Figure 4, the application has a video player on the right, and a scrolling display of annotations on the left. There is a button to control play vs. pause of the video player, and a button to control forward vs. reverse playback. There is also a horizontal slider showing time—it indicates the current position of playback within the video, and it can be used to reposition playback.

Video playback is synchronized with annotation display. Each annotation in the supplied sequence of annotations has an associated time interval. When video playback is within the time interval of an annotation, that annotation is positioned at the center of the display. If a user clicks on an annotation, playback is repositioned to a time within the interval associated with that annotation.

5. Language Design Issues

A high-level overview of the design of the application is shown in Figure 5. No events are shown at this level.

5.1 Bidirectional constraints

The constraints in our language are non-directional: two variables related by a constraint can influence each other, depending on which variable has been set by a an event. In contrast, both dataflow programming and JavaScript's event-drive programming are directional, akin to an assignment. As a result, if bidirectional influence is desired, these programming models will have to express the two flows separately, which may leads to larger programs and inconsistencies in semantics between the two directions.

The benefit of non-directionality of constraints shows up in several places in Figure 5.

Most importantly, time can be set by the AnnDisplay component, the VideoPlayer component, and the Slider. Each component can influence the others.

In addition, one of the ToggleBtn components changes the play versus pause state of the player. In the other direction along the constraint, when the player finishes the video (in either playback direction) it will set its play variable to false, and that will result in the button changing state appropriately.

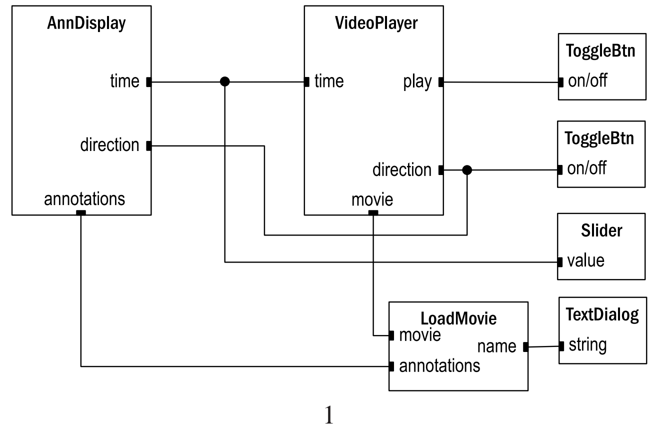

Figure 5. High-level view of the video player application

When a video name is entered into the TextDialog, the result is that a video is loaded into the VideoPlayer and an annotation sequence is loaded into the AnnDisplay. On the other hand, when the VideoPlayer initiates playback of a different video (functionality not shown in this figure), the TextDialog will update with the name of the video, and a corresponding annotation sequence, if one exists, is loaded into the AnnDisplay.

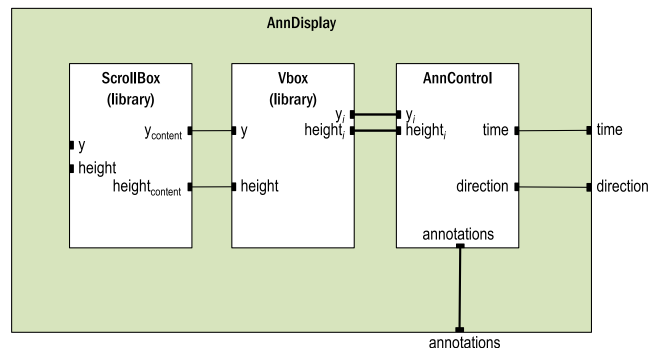

Figure 6. The layout of the annotations. The ScrollBox element displays a visible part of the VBox, which is a document container that stacks individual annotation elements, created in AnnControl.

Figure 6 shows what is encapsulated within the AnnDisplay component. This component specifies the layout of annotations. The crucial subcomponent is AnnControl, which creates one annotation text box for each annotation from the sequence of annotations. These boxes are not laid out in any way by AnnControl—it does not constrain their coordinates. The layout will be done by the Vbox component, which will stack these annotation boxes vertically. The ScrollBox will then take the Vbox and display it in a viewport to which a scrollbar is attached. The ScrollBox and the Vbox come from a library, but they too are written in the constraint language.

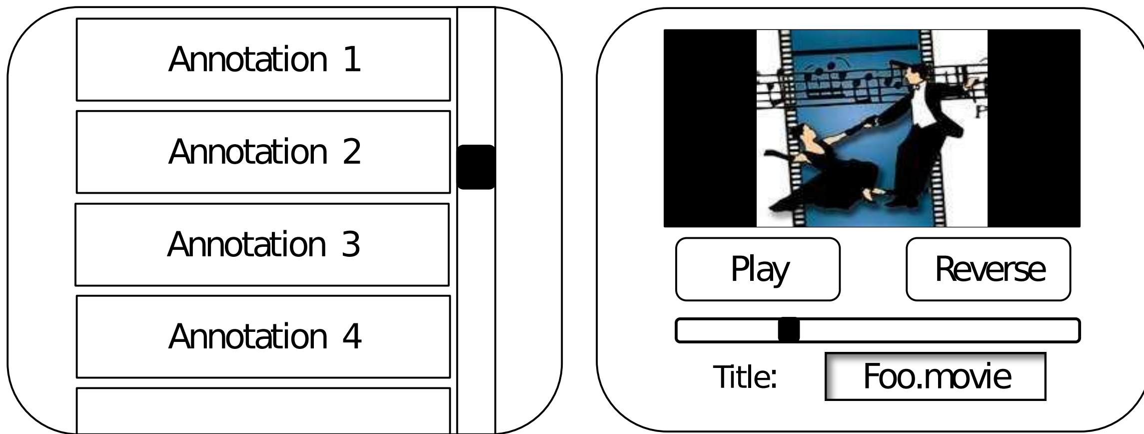

Figure 4. The media player application.

5.2 Directionality

When we desire directional influences between variables, they can be specified, too, using implications.

Consider the AnnControl component, in Figure 7, which is responsible for moving the active annotation into the viewport, as well as setting the time when an annotation is selected by a mouse click. In the figure, the green box contains a map combinator. The blue box is the component applied on each element of the list of annotations. The result is the list of Boxes, one per annotation. In the blue box, the split each annotation, which is a pair, into the text and the time interval. The text is then used to create a box. The time interval is used to establish a constraint that an active annotation is one whose internal contains the current time. So when the player changed the time, this constraint makes appropriate annotations active. In the opposite direction, on a mouse click, an annotation is made active, which changes the global time. Note that we elide the width and the x coordinate.

In AnnControl, we desire a directional dependence between active and the value of y. When an annotation is active, the coordinate y must be zero, which has the effect of scrolling the annotation to the top of the viewport. However, we do not wish a dependence in the opposite direction: when the user scrolls the annotations, he may set the value of y for some annotation to zero, but this should not influence the value of active because that would in turn influence the value of time; we wish to change the time of the playback only when an annotation box is clicked.

5.3 Ambiguous programs

An *ambiguous* program contains a constraint system that permits multiple solutions under some sequence of events. Such a program is nondeterminate and it is the task of the programmer to modify the constraint system to allow a unique solution under all sequences of events. We plan to use a model checker for detecting whether such a sequence of events exists. We do not discuss the model checker in this paper. Instead, on a linguistic construct for strengthening the ambiguous constraints system.

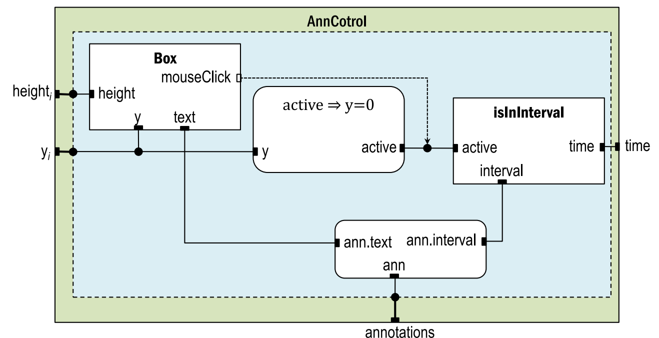

Figure 7. The annotation control module, AnnControl.

Consider AnnControl in Figure 7. When a mouse clicks on a box, the value of active becomes true, which forces the value of time to be within time interval of the clicked annotation. There is, however, ambiguity as to which time from the interval to select. The constraints system, therefore, is ambiguous.

Which allow the programmer to state the following constraint. The constraint says that at the moment when active becomes true, the time must equal either the start or the end of the interval, depending on the direction in which the movie is playing. Note that these constraints hold when active transitions from false to true; one can think of whenever is sending an event at this moment, and that event asserts a constraint on the value of time.

```
whenever active becomes true then
        if direction = forward then time = interval.start
        else time = interval.end
```

5.4 Inconsistent programs

An *inconsistent* program contains a constraint system that has no solution. The program may enter an inconsistent state only after a particular sequence of events. This subsection

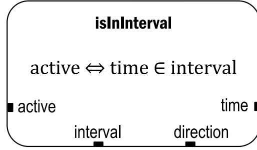

Figure 8. A constraint module that permits multiple solutions.

focuses on how the programmer modifies the constraint system to remove the inconsistency. Our approach is to relax the constraints and ask the constraints solver for an an optimal solution.

To show the inconsistency in our application, let us first describe how physical layouts are defined as constraints. Figure 9 shows a constraint module VBox that stacks display elements on top of each other. Logically, VBox wraps a list of display elements provided by another component in a new display element. It computes the height of the new element and places constraints on the positions (y_i) of the stacked elements. The positions are a function of the element heights. The actual positions of these elements depend on the placement of the VBox in its parent.

Recall that integrating layout and reactivity is one of the goals (in today's browsers, the context switch between the active scripting and layout is very expensive). Figure 9 shows that the layout style is under the programmer control and the layout constraints are indistinguishable from those constraints that react to events.

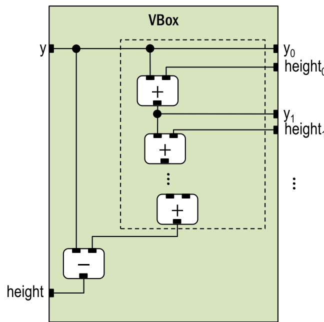

Figure 9. A VBox layout container contains a set of constraints that vertically stacks boxes attached on the right.

We are now ready to illustrate a scenario in which our program reaches an inconsistent state. Consider again AnnControl in Figure 7, which links the position of an annotation box with the current video time. When AnnControl, which creates annotation boxes, is composed with VBox, which lays them out, the programm has no solution when multiple annotations are active at the same time. If two annotations overlap in time, they both want to be displayed at the top of the scroll box, which is demanded by constraints $y_i = 0$ and $y_j = 0$, for some $i < j$. These constraints must hold in conjunction with the layout constraints in VBox, which demand $y_j = y_i + height_i + \ldots + height_{j-1}$. These constraints have no solution (unless $height_i + \ldots + height_{j-1} = 0$).

The programmer could remove the inconsistency by adding constraints that arbitrate among active annotations, allowing at most one annotation to specify its absolute position. This solution, however, requires adding constraints *among* the annotations, which would require changing, in Figure 7, the *map* operation, whose annotation constraints are independent per annotation box, into a more complex *fold* operation that would build a network of constraints that tie annotation constraints together, increasing program complexity.

Instead, we propose to use a constraint system computes an optimal solution subject to specified constraints. Such a constraint system may allow us to removed the inconsistency by relaxing the constraints; that unique solution will then be defined as the best solution. In our case study, this approach changes the constraint $active \Rightarrow y_i = 0$ to the constraint $active \Rightarrow$ (minimize y such that $y \geq 0$) (see Figure 10). When two annotations are active, the y coordinate of only one will be set to zero, which removes the consistency. In our future work, we will explore more case studies, to see whether optimization constraints simplify the task of creating consistent programs.

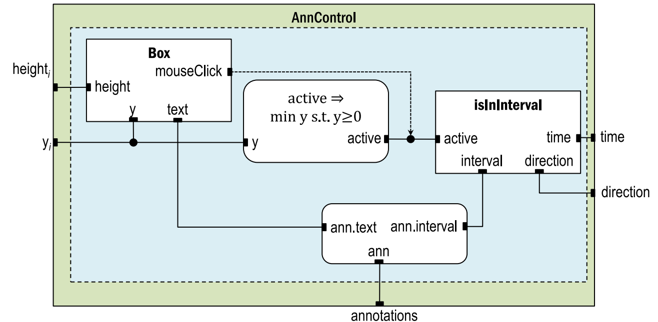

Figure 10. Revised AnnControl. To avoid contradicting constraints, the y coordinate is minimized rather than set to 0.

5.5 Programming by Demonstration

Declarative programming states *what* should be computed; it abstracts away from *how* the program computes the values that meet the specification. This very property, usually desirable, may make declarative programming harder than an "operational" one because constraints may be difficult to debug. Indeed, programmers may sometimes prefer to show the steps of the computation. The scrollbox component in Figure 11 may be one such example. This component includes two subcomponents that specify the size of the scrollbar, and the position of the content box within the viewport. These are mere linear constraints but they are somewhat difficult to write correctly. We think that meta-programming by demonstration could simplify constraint programming: if the programmer demonstrated the result of the program on

a sufficient set of corner case examples, the constraints can be inferred from these examples, which constrain the constraints.

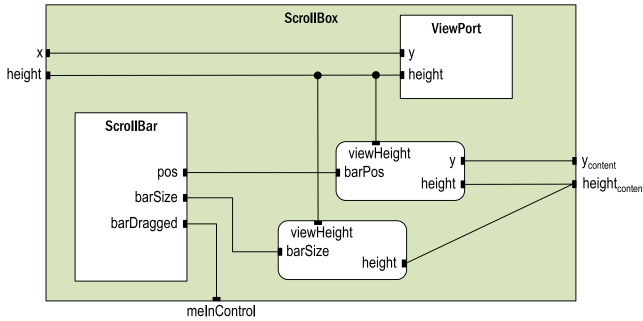

Figure 11. The scroll box container. Displays another box within a viewport. The two (linear) constraints specifying the position and size of the scroll bar are somewhat tedious to write and could be solved with user demonstration.

5.6 Abstraction (modularity)

The component-based architecture of the language makes it very easy for the programmer to identify the modules and their interdependencies in a program. Furthermore, it naturally offers several levels of abstraction to help the programmer. A component and its interactions with others can be analyzed by treating all the components as black boxes (c.f. Figure 5). Moreover, the subtle constraints restricting the ports of a component can be understood by looking at its inner content while abstracting the rest of the world away (c.f. Figure 9).

To build libraries of reusable components, a subtyping relation between components may need to be defined. We are currently working on a suitable definition of subtyping; should a subtype have more or less behaviors than its supertype?

5.7 Temporal Constraints

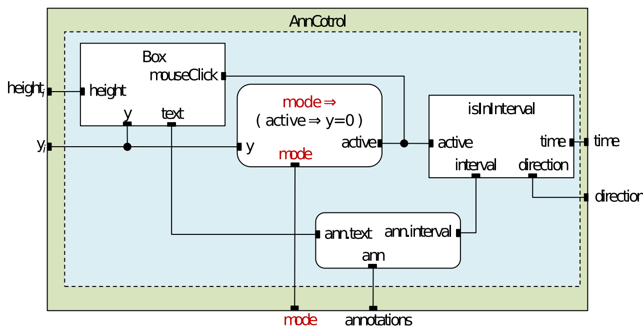

Figure 12. The annotation control module, AnnControl with two scolling modes.

There is no need to extend the language to express temporal constraints. In fact, they can be express with events if we allow components to fire events. For instance, in the movie player application, the current annotation to be displayed can be dictated by both the player itself and the user through the use of the scrollbar. For this reason, the annotation controller has two modes: one when the player is in control and another one when the scrollbar is (c.f. Figure 12). The scrollbar relinquishes the control of the annotations when the user releases the mouse from scrollbar; we do not wish to switch the mode back to the player immediately, though, as that would instantly scroll the annotations, creating a surprising effect. Instead, we want to delay the switch until the next annotation becomes active. We delay this action using an event that fires when any annotation becomes active. SO, firign an event on a transition achieves the desired temporal constraint.

6. Parallelism

In this section we discuss how we plan implement parallel constraint solving. We explain why a constraints-based language may simplify parallelization of the layout computation. The discussion here is informed by our experience with parallelizing a web browser [9]. As part of the project, we have been working on parallel algorithms for webpage layout [10]. We have found the parallelization task difficult for two reasons. First, the CSS layout constraints are specified informally [4] and are at times ambiguous. Second, the web page layout constraints are an instance of a *flow* layout style, which lays out a document element after the previous comments have been positioned. A flow layout performs an inorder tree traversal and is therefore inherently sequential. We were able to parallelize the flow layout by isolating five consecutive layout phases, each of which performs either preorder or postorder a tree traversal and thus exposes an opportunity for parallelization.

The technical challenge is to automate this decomposition of an inorder traversal of the document tree into a sequence of parallel tree traversals. That is, given a tree constraint structure, which subtrees can be solved independent of its siblings? The problem is made harder by the bidirectional nature of layout constraints: sometimes, the height of a document element is computed from its subelements, while sometimes the maximal height of a parent element is given and the height or margins of its subelements are computed so that they are spread equally within the parent element.

To identify parallelism, we plan to exploit the hierarchical structure of the constraint system. Figure 6 shows a part of the hierarchical document structure: the ScrollBox is a parent of the VBox, which in turn contains the individual elements, one per annotation, created in AnnControl. We hope to design component signatures that will inform the parallelizer of dependences among port components. For example, if the VBox component is informed that the value of $height_i$ does not depend on the value of y_i, then the chain of additions in VBox can be performed with a parallel adder tree.

7. Open Problems and Related Work

The essential features of the language have been designed but a few important extensions are still on the table. For example, we plan to design a dedicated sublanguage for animation and provide safe features for asynchronous interaction with the server [7]. Also, several features needed to make the language practical are missing. Finally, we are the first to admit that, in addition to more design work, a lot is left to be understood about the implications of the design choices we have already made. This section outlines these remaining questions.

The most interesting are the less apparent implications of mixing events with constraints will have to be uncovered. There have been a fair amount of work in extending constraint system with reactivity, notably [6] and [12]. It is not clear to the authors whether the methods presented in these two papers for finding solutions are applicable to our language. In tccp [12], the handling of one event produces the constraint system for the next event. We have instead chosen to (largely) fix the constraint system and update its solution minimally in response to an event. We suppose that one can convert from our minimal-update semantics to that of tccp, but it is unclear to us how small and natural the conversion is. It is also interesting to consider whether our minimal-update semantics is a good fit for all programs from the domain of browser programs. We expect that programmers may want to design custom update policies, and we do not know how difficult it will be for the programmer to ensure that there is always a unique minimal update.

Furthermore, cycles in the constraints network will make the task of finding solutions much harder. In fact, if cycles are allowed, it is possible to encode latches. Thus, it is no longer possible to describe a component as a relation since its inner latches can store state (see the discussion of hidden variables in Section 3). We will have to investigate whether the client side of a web application can be entirely stateless. Clearly, browser programs will need state, for example to cache email messages in an email client application. However, it may be possible to store the cache in a client-side database and access it from the program as if it was on a server, keeping the program stateless. Finally, we have conveniently ignored a question when presenting the language semantics: ”what is the logic of constraints?” It seems that *layout* needs conditional linear constraints but what are the constraints needed by the scripting logic?

Finding an efficient way to find solution to the constraint networks will be challenging. We hope the hierarchical nature of such networks will enables us extract as much parallelism as possible. State-of-the-art SMT solvers [2] have made tremendous progress the last five years; we hope to take advantage of them. Moreover, CSP ([14]) solvers have also improved by reusing the techniques developed for SMT solvers (c.f. [5]).

To enable reuse of code, components will have to be more modular. Convenient and easy to use component libraries might require more language support than we currently have. In particular, we will have to investigate whether a sensible subtyping relation between components can be defined in term of behaviors.

Finally, even though we handle events atomically, races may arise at a higher level of abstraction. As we show in [7], both interactions with the server and animations may produce races. In the former case, this happens when there are multiple outstanding requests to the server and the responses arrive out of order. We hope to design linguistic support that enables out-of-order messages (for performance reasons) but allows the programmer to specify how to handle messages that arrive early. This specification should be given in terms of the application semantics, not in the terms of low-level messages timestamps. For example, to handle two back-to-back user requests to zoom into a map, the programmer may want to state that the screen is allowed to display a stale map tile (one corresponding to the request before the last one) because seeing at a map at a wrong zoom level is better than waiting for the correct map to appear.

8. Conclusion

This paper describes a preliminary design of a language for web browser programming. Our goal is to integrate scripting and page layout, while raising the level of abstraction of JavaScript and HTML/CSS — the respective solutions for the two problems — and simultaneously improving their performance. Our design is motivated by limitations of today's browser programming (as reported to us by programmers and browser developers) and by some experience with functional reactive programming in the browser (we had our undergraduates implement and program with Flapjax [11]). We believe that we have a good understanding of the domain and we describe an approach for combining constraints and events that appears natural for this domain. This paper also describes how inconsistent and ambiguous constraint systems may arise in this domain, and we suggest how programmers may deal with them with the proposed language. A lot remains to be designed and understood. Section 7 lists the known unknowns, and we hope that the discussion at the workshop will uncover the unknown unknowns [13].

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.

[2] Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.

[3] Ras Bodik, Justin Bonnar, and Doug Kimelman. Productivity programming for future computers. *FIT*, 2009.

[4] Bert Bos, Tantek Celik, Ian Hickson, and Hakon Wium Lie. Cascading style sheets level 2 revision 1 (css 2.1) specification. http://www.w3.org/TR/CSS2/, 2009.

[5] Ian P. Gent, Chris Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *In: Proceedings of ECAI 2006, Riva del Garda*, pages 98–102. IOS Press, 2006.

[6] Pascal Van Hentenryck. Scheduling and packing in the constraint language cc(fd). In *IN INTELLIGENT SCHEDULING. ZWEBEN AND FOX (EDS)*. Morgan Kaufmann, 1994.

[7] James Ide, Rastislav Bodik, and Doug Kimelman. Concurrency concerns in rich internet applications. In *Exploiting Concurrency Efficiently and Correctly (EC2), CAV 2009 Workshop*, 2009.

[8] Radha Jagadeesan, Will Marrero, Corin Pitcher, and Vijay A. Saraswat. Timed constraint programming: a declarative approach to usage control. In Pedro Barahona and Amy P. Felty, editors, *PPDP*, pages 164–175. ACM, 2005.

[9] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic, and Rastislav Bodik. Parallelizing the web browser. In *First USENIX Workshop on Hot Topics in Parallelism (HotPar '09)*, 2009.

[10] Leo Meyerovich. Fast webpage layout. http://pbrowser.googlecode.com/svn/trunk/layout/match/writeup/rulematching.pdf, 2009.

[11] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for ajax applications. In *The International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2009)*, 2009.

[12] Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Foundations of timed concurrent constraint programming. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80. IEEE Computer Press, 1994.

[13] Hart Seely. The poetry of D.H. Rumsfeld. http://www.slate.com/id/2081042/, 2003.

[14] Roland H. C. Yap. Constraint processing by rina dechter, morgan kaufmann publishers, 2003, hard cover: Isbn 1-55860-890-7, xx + 481 pages. *Theory Pract. Log. Program.*, 4(5-6):755–757, 2004.