Wallingford: Toward a Constraint Reactive Programming Language

Alan Borning
University of Washington, USA and Communications Design Group
borning@cs.washington.edu

Abstract

Wallingford is an experimental constraint reactive programming language that supports objects with state, persistent constraints on those objects, and reactive constraints that enable the system to respond to events by declaring constraints that become active when an event occurs, or while a condition holds. It is implemented in Rosette, a solver-aided programming language and environment for constructing DSLs in turn implemented in Racket. This short paper is a snapshot of work-in-progress on the system.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Constraints

Keywords Object constraint languages, reactive programming, Rosette

1. Introduction

Wallingford is an experimental constraint reactive programming language, which integrates constraints with reactive programming, in a way that cleanly accommodates state and change, and that at the same time favors being declarative rather than imperative when feasible. The initial target applications are interactive graphical ones that use object-oriented features, and constraints to specify relations among the graphical elements and also how the application reacts to inputs. The language draws on previous work on Babelsberg (Felgentreff et al. 2014, 2015b), a family of object constraint languages, and is implemented in Rosette (Torlak and Bodik 2013, 2014), a solver-aided language that extends Racket (a Scheme dialect).

This position paper is a snapshot of the work to date. Section 2 discusses three key influences on the design of Wallingford, namely Babelsberg, Rosette, and the Fran functional reactive programming system (Elliott and Hudak 1997). Section 3 describes a small set of macros and methods that extend Rosette for use in representing objects with state and persistent constraints, while Section 4 discusses adding reactive constraints.

A prototype implementation of the core Wallingford system described in Section 3 is complete and reasonable stable. The additions to support reactive constraints (Section 4) are being implemented, with some but not all being currently operational. The current system supports some simple interactive demonstrations, but performance is explicitly not a goal at this time. Rather, the goal is to work out a clean and expressive language; at the same time, the language should not include features that would make it impossible to develop much faster implementations later. All code is available in the repository `https://github.com/cdglabs/wallingford.git`

To appear, Proceedings of the Constrained and Reactive Objects Workshop (CROW'16), March 15, 2016, Málaga, Spain

2. Related Work

Arguably the most common approach to integrating constraints with a programming language is to extend logic programming, for example as is done in the constraint logic programming scheme (Jaffar and Lassez 1987), or by using integrated constraint libraries as in SWI Prolog. There are a number of significant advantages to this approach, including a generally clean semantics and programming model. However, it does not provide state or reactivity, which are two goals for Wallingford. Instead, Wallingford draws more directly on languages and systems that do provide state and reactivity. This section describes three: the Babelsberg family of object constraint languages, the Rosette solver-aided language, and the functional reactive animation system Fran.

2.1 Babelsberg

Babelsberg is a family of object constraint languages that allows constraints to be integrated with an existing object-oriented language. Current implementations are Babelsberg/Ruby, Babelsberg/Javascript, and Babelsberg/Squeak (Squeak being a Smalltalk dialect).

Babelsberg enables the programmer to write constraints on objects that include OO constructs such as message sends and that respect object encapsulation. For example, this constraint specifies that the center of a rectangle `r` remain at a fixed position as its other attributes are updated.[1]

```
always r.center() = Point(10,20)
```

It doesn't matter whether `center` is stored as a field of the rectangle or computed via a method; in either case, Babelsberg will evaluate the constraint expression in constraint construction mode to rewrite it into a set of primitive constraints that can then be turned over to the solver.

A key aspect of Babelsberg is the handling of state, assignment, and object identity (Felgentreff et al. 2015b). Babelsberg makes a distinction between instances of primitive types and of value classes, for which object identity is not significant and that don't need to be stored on the heap, and instances of classes for which

[1] The examples in this section are written in pseudo-code, following the examples in (Felgentreff et al. 2015a).

identity is significant and that are stored on the heap. Here is a simple example using just primitive types.

```
x := 3;
y := x;
always y=10;
```

At the end of this fragment we have `x=3` and `y=10`. `x` and `y` hold instances of a primitive type, and so even though we assign `x` to `y`, they are not aliased, and the `always` constraint only affects `y`. In addition to explicit constraints (such as `y=10` above), there are implicit soft stay constraints to give stability in the absence of other stronger constraints — so after the initial assignment to `x`, there is a low-priority stay constraint that it retain the value 3. There is also a low-priority stay constraint on `y` after the assignment `y:=x`, but this stay is overridden by the required constraint `y=10`.

On the other hand, for ordinary objects that aren't primitive types or instances of value classes, object identity *is* significant. Aliases can be either implicitly created using an assignment statement, or by using explicit identity constraints. Reference (Felgentreff et al. 2015b) discusses Babelsberg's approach to taming the power of the constraint solver so that it can't on its own create new objects with object identity to satisfy constraints, add fields or methods to objects, or the like.

Semantically, all the constraints are solved after each statement. In implementations, however, Babelsberg uses various techniques to enable reasonable performance, such as keeping track of dependencies so that only constraints involving changed objects need be re-solved, and "edit" constraints to take advantage of incremental solvers.

2.2 Rosette

Rosette is a solver-aided language that extends Racket (Racket 2016) with symbolic data types and solver operations. The motivation for using Rosette as an implementation platform for Wallingford are two-fold. First, Rosette includes support for inferring constraints and programs from examples, and this capability will very likely be useful in the future for automatically inferring graphical constraints from example layouts. Second, it is a thought-provoking system, and I simply have an intuition that interesting results will flow from trying to meld it with the Babelsberg approach to objects and constraints (for example, by using Rosette's symbolic variables). The Rosette system supports a variety of constraint solvers — the Wallingford implementation uses Z3 (De Moura and Bjørner 2008).

As a simple Rosette example, the following statements declare two symbolic variables and constraints on them, and then find a solution that satisfies these constraints. (The Rosette `solve` macro takes an expression that can assert additional constraints; since we aren't using that capability here we just use `#t`.)

```
(define-symbolic x y number?)
(assert (equal? y (+ x 2)))
(assert (equal? y 10))
(solve #t)
```

The previous example of a constraint on the center of a rectangle has the following analog in Rosette:

```
(assert (equal? (rect-center r) (point 10 20)))
```

As with Babelsberg, the expression is evaluated in the host language, and `(rect-center r)` will be partially evaluated away prior to reaching constructs known to Rosette. However, the `assert` statement just asserts the constraint at the time it is evaluated, unlike `always` in Babelsberg that re-asserts the constraint at each time step.

Such persistent constraints declared using `always`, as well as the low-priority stay constraints, are not supported primitively in Rosette, but it is straightforward to add them in a DSL on top of Rosette, as described in the next section.

An important difference between Babelsberg and Rosette concerns the handling of variables, state, and assignment for immutable types. In Babelsberg, for a variable that refers to an immutable object, a weak stay constraint causes it to retain its value in the absence of an explicit assignment or stronger constraints. The value itself is however immutable. Concrete values in Rosette are similar; but to make use of constraints, programmers declare symbolic constants. They can then assert constraints on these symbolic constants and ask the solver for a satisfying solution. Programmers can also make aliases for these symbolic constants, or pass them to procedures and in those procedures put assertions on the symbolic constants. In contrast, in Babelsberg, aliasing doesn't apply to instances of primitive types. In Rosette, transparent immutable structures can hold symbolic constants, and so the same considerations apply to them as to symbolic constants themselves.

Thus, in Babelsberg, types without object identity are immutable — constraints on variables and stay constraints are used to update the values as time progresses. In Rosette, symbolic constants, and immutable transparent structures that contain them, form a kind of shell. The programmer can clear all the assertions on them, or even make assertions that apply only within a particular call to `solve`; but the structure persists independent of these assertions. This concept of symbolic values and different bindings does not exist in Babelsberg. On the one hand, the Rosette approach bypasses the issues with taming the power of the constraint solver that arise in Babelsberg, since the solver just works on the leaves (symbolic numbers, booleans, etc), and not on user-defined structures. On the other hand, the situation with ordinary objects in Babelsberg, and mutable or opaque structs in Rosette, is much closer. In both systems, the solver can't make new objects or spontaneously change the object to which a variable refers to satisfy constraints.

2.3 Fran

There has been substantial work on reactive programming languages and frameworks — see reference (Bainomugisha et al. 2012) for a useful survey. An early and influential system is Fran by Conal Elliott and Paul Hudak, and it has been an important inspiration for the work on adding reactive constraints (Section 4). Fran provides an integrated set of data types and functions in the Haskell language designed for writing animations, in particular *behaviors* and *events*. As is implied by being written in Haskell, it is a purely functional system, and graphical elements and other behaviors are implicitly functions of time rather than having state that changes with time. In Fran, time is continuous, not discrete. *Interval analysis* (Snyder 1992) is used to perform event detection.

Typically, imperative reactive languages, for example Flapjax (Meyerovich et al. 2009), use behaviors and events as well. In all of these languages, functions or methods must then be *lifted* to operate on behaviors rather than ordinary values.

3. The Core Wallingford System

The core Wallingford system includes a small set of macros and methods that extend Rosette for use in constructing interactive graphical applications, as well as other applications involving objects with state and persistent constraints. The key features provided by the core system are facilities for building objects with persistent constraints, such as constrained geometric objects, GUI widgets with constraints, or simulation elements; and for implementing frame axioms (specifying that things remain the same unless there is some reason they should change). Soft constraints as well as hard (required) ones are useful for both of these features.

In particular, the frame axioms are implemented using low-priority "stay" constraints that instruct the solver to leave the value of a variable as it was, unless forced to change by some stronger constraint.

These features are provided by a class `thing%` that includes the following macros and methods. (The Racket convention is that class names end in `%`.)

`always` This macro takes a constraint and an optional priority, and adds that constraint to the set of constraints that will be enforced thereafter. The priority defaults to `required`. For example, here is a constraint to keep a midpoint halfway between the two endpoints of a line:

```
(always (equal? midpoint
  (point-times 0.5
    (point-plus (line-end1 line1)
                (line-end2 line1))))
```

`always*` This provides similar functionality to `always`, except that the expression is re-evaluated at each step, so that the constraint applies to the new bindings of variables if they are re-assigned.

`stay` This macro takes an expression and a priority (which defaults to "lowest"), and adds a stay constraint to the system that the value of the expression stay the same each iteration. For example, this constraint tells the system that one of the endpoints of a line should not move unless necessary, with low priority:

```
(stay (line-end1 line1) #:priority low)
```

`clear` This method cleans out all of the constraints and stays owned by a particular thing.

`solve` This method is similar to Rosette's `solve` macro, augmented with features to support Wallingford. It finds and returns a solution to the explicit assertions plus all constraints declared using `always` and `stay`. Stay constraints are considered relative to the current state of the thing. The solution returned from `solve` can then be used to evaluate an expression involving symbolic variables to find a concrete value. The system uses iterative deepening to find a solution to the constraints that satisfies as many of the soft constraints as possible, respecting their relative priorities. (Using the terminology of (Borning et al. 1992), specifically it finds a unsatisfied-count-better solution.)

4. Adding Reactive Constraints

This section describes work in progress on adding temporal constraints and reactivity to Wallingford. Many of the examples and features are inspired by Fran (Elliott and Hudak 1997), although, as a language with state, in many ways it is not at all like Fran. While the current design is being developed using Rosette and the framework presented in the previous section, the same ideas could be used in other constraint languages as well.

Reactivity is encapsulated in a class `reactive-thing%`, which is a subclass of `thing%`. Each reactive thing has its own internal time — the plan is for time to be continuous, as in Fran, but in the current implementation it is an integer (milliseconds since system start time), due to Rosette limitations.[2]

[2] This limitation should be removed in the near future. Z3 itself does provide real numbers, so this is a matter of making this functionality available in Rosette. Note that these are mathematical reals, not floats. (Again, the goal at this point is getting a clean and expressive language, rather than performance — there will be many opportunities for optimization later.)

Reactive things have a default image, and we can set up a *viewer* that periodically samples the state of the reactive thing and displays it. This sampling can be done either on a pull or a push basis — the semantics are that of pulling (i.e., periodic polling) — push notification is an optimization that doesn't change the semantics. Reactive things understand the message `get-sampling`, which returns one of `push`, `pull`, `push-pull`, or `none` (for static things). The recommended kind of sampling is determined automatically by analyzing the thing's set of constraints. Each reactive thing, as well as each viewer, has its own thread, implemented using the Racket thread mechanism, and communicate as needed. For example, if pull sampling is being used, the viewer will periodically poll the viewed thing to ask it to advance time and then to show its current image; while if push is being used, the thing will notify the viewer that something potentially worth observing has occurred.

Constraints on reactive things can reference the thing's current time, making them temporally dependent. For example, we can create a reactive thing that includes a blue circle `c` as its image, and make the circle smoothly change size as a function of time:

```
(always* (equal? (circle-radius c)
                 (+ 60 (* 50 (sin (seconds))))))
```

In this case, there is a pure pull relationship between the viewer and the viewed thing — viewer polls the viewed thing as often as need be to get a smooth animation. (In the current implementation, this example runs with some jerkiness, since the system is actually calling Z3, as an separate process, to solve the constraint on each refresh.) To do this, the viewer sends the reactive thing a message `advance-time` with a proposed new time t as an argument. The reactive thing responds by advancing its internal clock appropriately; and then its image can be retrieved and shown using the new time t. Semantically, the viewed thing takes on all of the times between its current time and t, but since there are no detectable effects of taking on all the intermediate states, the thing's internal time can simply jump to t.

Conditionally asserted constraints are supported using `when` and `while` macros.

when takes a boolean-valued expression and asserts a set of constraints at the instant that event occurs.

while similarly takes a boolean-valued expression, and asserts a set of constraints that should hold as long as the condition has the value true.

Both `when` and `while` have a boolean-valued expression as their conditions, but for `when` the expression is restricted to denoting an *event* that is true just for an instant.[3] The default duration for a `while` or `when` macro is `always`: it should hold from that point onward. These macros also allow for an `until` clause with an event-valued expression, which specifies that the `while` or `when` should be deactivated if the event occurs.

For example, we can add a `when` temporal constraint to our circle to make it flip between blue and red every 5 seconds:

```
(define (flip c)
  (if (equal? c (color "blue"))
      (color "red")
      (color "blue")))

(when (zero? (remainder (milliseconds) 5000))
  ((equal? (colored-circle-color c)
     (flip (previous (colored-circle-color c))))))
```

[3] We can formalize this restriction as follows. For any time t such that the condition c of the `when` has the value true, there exists some $\delta > 0$ such that c is false for all other times in the interval $[t - \delta, t + \delta]$.

If a reactive thing includes `when` or `while` temporal constraints, handling the `advance-time` message is more complex, since a `when` condition might be true at some instant between its current time and the new proposed time, or a `while` condition might change from false to true or vice versa. Even in the current implementation, in which time is represented as an integer, it would be quite inefficient to methodically advance time 1 millisecond per tick, checking each tick for `when` conditions that have become true or `while` conditions that change value; and impossible when the implementation is rewritten to use real-valued time. Instead, the system uses its constraint solving capabilities to find the next time to which to advance. Let the reactive thing's current time be r, and the new proposed time be t. The system then solves for a time s such that $r < s < t$, all of the required constraints hold, and at least one condition in a `when` holds or a `while` condition changes its value. If no such time exists, the reactive thing can safely advance its time to t. But if such a time s does exist, the system uses iterative deepening to find the minimum such time $s_{\min}$, and the thing advances its clock to $s_{\min}$. It then uses the Wallingford version of `solve` to solve the constraints at $s_{\min}$ (including the soft constraints), asserts the constraints in the bodies of all `when` or `while` statements whose condition holds at $s_{\min}$ (for a `when`, just at that instant, and for a `while`, as long as the condition holds — like a bounded version of `always`), and finally calls `solve` again. Finally, if there were such a time $s_{\min} < t$, the reactive thing calls `advance-time` again to attempt to advance its time to t. (It might need to advance to multiple intermediate times before eventually getting to t.)

The function `previous` used above denotes the value of the circle's color at some time p less than the current time and greater than any other time that the thing's state has been accessed. (Thus, `previous` can only be used in a `when` constraint — it doesn't make sense in a `while`.)

Incidentally, the condition in the above `when` takes advantage of milliseconds being an integer — when this is converted to use reals, the condition would need to also include a test `(integer (milliseconds))`.

External events can also be accommodated in this framework. For example, the function `left-button-pressed` returns true just at those instants that the button is pressed. (Note that this is not the same as `left-button-down`, which returns true throughout the interval the button is down — the "pressed" function returns true just at one instant.) We can then use such functions in `when` conditions, for example to flip the color when the button is pressed:

```
(when (left-button-pressed)
  ((equal? (colored-circle-color c)
    (flip (previous (colored-circle-color c))))))
```

If the reactive thing is monitoring the button state, then it maintains a list of times that the button is pressed; and when advancing time, we need to advance to the first time such that a condition on a `when` is true or the condition on a `while` changes value, including ones involving these external events. For example, suppose that a reactive thing `r` includes the above constraint, and there is a button press at time 2000 (milliseconds). If `r` gets an `advance-time` message asking it to advance to 1000, it can simply do so.

On the other hand, if it is asked to advance time to 3000, then that method will find that the `when` condition on the flip constraint is satisfied at some earlier time, so it will just advance to 2000 instead. The constraints in the body of the `when` will be satisfied, and it will try again to advance to 3000, and assuming there is no intervening button press, it can do so.

As noted in Section 2.3, other reactive languages, both purely functional ones such as Fran, and imperative ones such as Flapjax, typically use separate types for behaviors and events, requiring that ordinary functions or methods be lifted to operate on them. In contrast, in Wallingford, in place of a behavior to represent for example a button down event, `left-button-pressed` is simply an ordinary function that returns true or false: the complexity is removed here, and instead occurs in the implementation of `advance-time`, which is provided by the underlying system and hence need only be written once by the Wallingford implementor. Thus, the hope is that the programmer will be presented with a much simpler model.

5. Future Work

This short paper is a snapshot of work-in-progress. As of this writing, the class `reactive-thing%`, along with `when` temporal constraints, the code to automatically determine the sampling scheme, and the class `viewer%`, plus unit tests and some example programs, is implemented and tested. Still remaining is implementing `while`. The next steps are thus to complete the prototype implementation of the reactive programming constructs and to test them on more complex interactive applications. Two additional temporal constraints are also planned: `(integral f)`, which denotes the definite integral $\int_{t_0}^{t_1} f\,dt$ for an expression f, where t_0 is the time at which the integral constraint is asserted and t_1 is the current time (modeled after an analogous construct in Fran); and `(detect expr)`, which takes a boolean-valued expression and returns an event that is true at each instant the value of `expr` changes to true.

Following that, getting reasonable performance through compilation and other techniques will be essential. Other future work will include developing a more formal semantics for the temporal constraints, using Rosette's capabilities to support inferring constraints from examples, and perhaps working out how Wallingford could function in a distributed system with multiple clocks.

Acknowledgments

Many thanks to colleagues in UW Computer Science & Engineering and the Communications Design Group, as well as to the CROW referees, for useful feedback on this work. This research was funded in part by SAP and Viewpoints Research Institute.

References

E. Bainomugisha, A. L. Carreton, T. V. Cutsem, S. Mostinckx, and W. D. Meuter. A survey on reactive programming. *ACM Computing Surveys*, 2012.

A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, Sept. 1992.

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

C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997. URL `http://conal.net/papers/icfp97/`.

T. Felgentreff, A. Borning, and R. Hirschfeld. Specifying and solving constraints on object behavior. *Journal of Object Technology*, 13(4):1–38, 2014.

T. Felgentreff, T. Millstein, and A. Borning. Developing a formal semantics for Babelsberg: A step-by-step approach. Technical Report 2014-002b, Viewpoints Research Institute, 2015a. Available at `http://www.vpri.org/pdf/tr2014002_babelsberg.pdf`.

T. Felgentreff, T. Millstein, A. Borning, and R. Hirschfeld. Checks and balances — constraint solving without surprises in object-constraint programming languages. In *OOPSLA*, Oct. 2015b.

J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the Fourteenth ACM Principles of Programming Languages Conference*, Munich, Jan. 1987.

L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for Ajax applications. In *OOPSLA*, 2009.

Racket. The Racket programming language, 2016. `http://racket-lang.org`.

J. M. Snyder. Interval analysis for computer graphics. *ACM SIGGRAPH Computer Graphics*, 26(2):121–130, 1992.

E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Onward!* ACM, 2013.

E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*. ACM, 2014.