Constraints as Polymorphic Connectors

Marcel Weiher[†,‡] Robert Hirschfeld[†,§]

[†] Hasso Plattner Institute, University of Potsdam, Germany
[‡] Microsoft, Germany
[§] Communications Design Group (CDG), SAP Labs, USA; Viewpoints Research Institute, USA
marcelw@microsoft.com robert.hirschfeld@hpi.de

Abstract

The architecture of interactive systems does not match the procedural decomposition mechanisms available in most programming languages, leading to architectural mismatch.

Constraint systems have been used only as black boxes for computing with primitive values of a specific type and for specific domains such as user interface layouts or program compilation.

We propose constraints as a general-purpose architectural connector for both describing the large-scale structure of interactive systems and matching that description with the actual implementation.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features— Constraints ; D.2.1 [*Software Architectures*]: Domain-specific architectures

Keywords Architecture;Connectors;Constraints;Polymorphism

1. Introduction

Interactive programs have been characterized as time-consuming and error-prone to produce [33] [34]. One of the reasons for this assessment is the architectural mismatch [24] between procedural programming languages and the decidedly non-procedural nature of interactive programs [13].

Due to the linguistic primitives available, programming languages carry with them a preferred architectural style, and strongly encourage programs written in that language to follow that architectural style. For popular programming languages used in the construction of interactive programs, such as Java, C#, Objective-C and JavaScript that is the *call-and-return* architectural style [44]: control flow is in the hand of the program, functions and methods are called and return results to their caller.

Interactive programs do not match this style; they typically do not compute results and do not return to their caller. Instead they maintain various invariants while reacting to input from their environment, including the user and often nowadays the network. Although these invariants can be and often are implemented procedurally, the overall structure of the program does not resemble a procedure.

MODULARITY'16, March 14–17, 2016, Málaga, Spain
ACM. 978-1-4503-3995-7/16/03...
http://dx.doi.org/10.1145/2889443.2889456

While working on various applications including a widely used task management app, we have found that the top level architecture can often best be described as a series of equality constraints maintained between the main components of the system. The user interface should reflect the state of the in-memory model as should the on-disk representation, and all of these should be synchronized with the data on other devices, including the user's own and other users' in case of shared data.

Maintaining invariants is a good fit for constraints, but constraints have so far been used primarily as computational components to solve domain specific problems such as user interface layout, logical puzzles or optimization problems.

In addition, constraint solvers have been constructed as black boxes acting on specific data types, limiting their use as structuring elements to specific solver implementations and data types.

Instead, we see constraints as a structuring mechanism for describing relationships between components, even when these are implemented without a black-box constraint solver, or even without any kind of identifiable solver. In short, they are architectural connectors.

```
1  memory-model  := persistence.
2  persistence   |= memory-model.
3  ui           =|= memory-model.
4  backend      =|= memory-model.
```

Listing 1. Wunderlist Client Architecture.

Listing 1 describes the high-level architecture of the Mac and iOS Wunderlist [32] clients:

- Line 1 initializes the memory model from the persistence layer using a high-level assignment operation (`:=`).
- Line 2 uses the one-way dataflow connector `|=` to keep the persistency layer in-sync with changes to the memory model.
- Line 3 uses the bi-directional dataflow connector `=|=` to keep the memory model and the UI in sync with each other.
- Finally, line 4 also uses the bi-directional dataflow connector `=|=` to keep the memory model in sync with the back-end.

Although we use only two dataflow connectors (one-way `|=` and two-way `=|=`) to describe the relationships between the Wunderlist components, the implementations of those connectors differ significantly depending on which components they connect and on the direction of the connection.

It should be noted that these are not all the components in the application, nor all the connections between components. Nor is it the only way to structure the application. However it is a useful decomposition that captures significant architectural elements and their interactions and would be recognizable to the developers working on the system.

We therefore propose treating constraints as a polymorphic architectural connector [42] [39], both in their use and in their internal construction. These constraint connectors are useful for describing the overall architecture of interactive systems like the collaborative task list manager shown in Listing 1, whether those top-level connectors are themselves implemented using constraints or not.

In the remainder of the paper we will first show the types of problems encountered in building interactive software with a brief example. Next we describe the constraint connector mechanism and a polymorphic treatment of constraint connectors by showing several useful variants and their application to the motivating example. We then show how those same mechanisms apply to the industrial example shown in Listing 1. Finally we present related work in software architecture and constraint programming and point out future avenues of research opened by this approach.

2. Motivating Example

As an example of the architectural issues faced when assembling even fairly straightforward interactive applications, we will look at a temperature converter application that converts between different temperature scales, starting with Fahrenheit and Celsius.

2.1 Objective-Smalltalk

All code presented here is expressed in Objective-Smalltalk [50], a Smalltalk [25] dialect. Objective-Smalltalk generalizes Smalltalk's support for object-oriented programming to support for defining and using architectural connectors and borrows method-definition syntax from Objective-C. It also generalizes identifiers to Polymorphic Identifiers, which look like URIs in code [52]. The goal is for these enhancements to allow Objective-Smalltalk to come close to a Domain Specific Language (DSL) [31] in expressiveness for many common tasks.

```
obj msg.                   // unary    obj.msg()
obj msg:7.                 // keyword  obj.msg(7)
3 * 4.                     // binary   3.*(4)
c := 7.                    // assignment
-c {                       // instance method
   ^ivar:c                 // return instance var c
}
ivar:field/value   // field.value;
```

Listing 2. Objective-Smalltalk Syntax

Listing 2 gives a quick overview of the Objective-Smalltalk syntax: an instance method of a class is introduced with the a minus sign, followed by the method signature and the method body enclosed in curly braces. Class methods would be introduced with the plus sign, but we don't have any class methods in these examples. Syntax inside methods is Smalltalk, with unary, binary and keyword messages, statements terminated with periods and method return indicated by the up caret.

Identifiers look like URIs, with a scheme separated from the path by a colon, so `ivar:c` is the variable `c` in the `ivar` (instance variable) scheme. Components of a path expression are separated by the slash character typical of file systems rather than the dots more typical of languages like Java.

2.2 Basic Model

The basic model of the temperature converter consists of storage for the temperature and methods to set and inquire that temperature in different temperature scales. An implementation of the model object is shown in Listing 3. To its clients it presents an interface with two properties, `c` and `f` representing the temperature in Celsius and Fahrenheit respectively. Internally, it stores the temperature in a hidden instance variable `c` and derives the Fahrenheit value on-demand, as well as converting Fahrenheit values on input.

```
- f:degreesF {
    self c:(degreesF - 32) / 1.8.
}
- f {
    ^ self c * 1.8 + 32.
}
- c:degreesC {
    ivar:c := degreesC.
}
- c {
    ^ivar:c.
}
```

Listing 3. Basic Temperature Converter Model

To keep the exposition manageable, we don't show boilerplate class definitions, instance variable definitions and generic application initialization code.

2.3 Connecting UI Elements

For the UI, we assume we have text fields set up to input and display numbers using Apple's Cocoa UI toolkit [3]. Our code for hooking up those text fields to the model is shown in Listing 4. The code for creating, positioning and configuring the text fields is not shown, because we are primarily interested in the connecting code.

Lines 1-7 contain target/action methods that are called by the text fields either when the user finishes editing or on every keystroke, depending on how the fields are configured. Each target/action method has a `sender` parameter that contains a reference to the control that sent the action message. In our case, we ask the control for its numeric value and set that as the corresponding temperature, either Fahrenheit or Celsius depending on the actual control.

So far we have only had straightforward additions, but in order to update the calculated values in the UI, we have to modify our existing setter methods, both the "virtual" setter for Fahrenheit that just computes a Celsius value and the actual setter for the Celsius value.

Each of these setters updates the other UI value, setting Celsius needs to update the Fahrenheit text field, but not the Celsius text field because that is presumably where the value originated.

```
1   - changedF:sender {
2       self f:sender intValue.
3   }
4   - changedC:sender {
5       self c:sender intValue.
6   }
7   - f:degreesF {
8       self c:(degreesF - 32) / 1.8.
9       ivar:ui/celsiusTextField/intValue := self c.
10  }
11  - f {
12      ^ self c * 1.8 + 32.
13  }
14  - c:newValue {
15      ivar:c := newValue.
16      ivar:ui/fahrenheitTextField/intValue := self f.
17  }
18  - c {
19      ^ivar:c.
20  }
```

Listing 4. Connecting UI via messages

While seemingly straightforward, the code in Listing 4 has a slight problem, at least if we are serious about minimizing UI updates: when setting Fahrenheit values, we call the Celsius setter after computing the correct temperature, meaning that we *do* redundantly update the UI with an already present value, at least in the case we convert Fahrenheit to Celsius.

Listing 5 fixes this problem by splitting the Celsius setter into two parts, one low-level part that doesn't do UI updates and is called when converting from Fahrenheit, and one high-level part that is called when actually entering Celsius for conversion and does do the UI update. (Only the methods that were changed are shown).

```

- f:degreesF {
    self basicC:(degreesF - 32) / 1.8.
    ivar:ui/celsiusTextField/intValue := self c.
}
- c:newValue {
    self basicC:newValue.
    ivar:ui/fahrenheitTextField/intValue := self f.
}
- basicC:newValue {
    ivar:c := newValue.
}
```

Listing 5. Minimizing UI updates

2.4 Adding Persistence

Adding persistence also requires making modifications to existing code, though only to a single method which is why Listing 6 only shows that single modified method. Whenever we set a new Celsius value, we write this value to the user defaults database.

```
- basicC:newValue {
    ivar:c := newValue.
    NSUserDefaults standardUserDefaults
                   setObject:newValue forKey:'c'.
    self updateUI.
}
```

Listing 6. Persistence using native API

As we can see, the cover setter method for the Celsius variable is getting to be a hub of changes for any additional architectural dependencies.

To make the code more comparable to the constraint solution we build in Section 4, Listing 7 shows the same code expressed with a Polymorphic Identifier using the `defaults` scheme instead of a message-send to the `NSUserDefaults` shared instance. The two pieces of code have the same semantic.

```
- basicC:newValue {
    ivar:c := newValue.
    defaults:c := ivar:c.
    self updateUI.
}
```

Listing 7. Persistence using Polymorphic Identifier

This is a very simple persistence solution, as we are only concerned about a single value, without any relationships, object identity or complex queries to worry about. In a more complex application, dealing with persistence is likely to be much more troublesome [36].

2.5 Adding a Temperature Scale

Adding a new temperature scale, in this case the Kelvin scale that starts at absolute zero, involves adding the conversion methods for the new scale, adding UI elements (not shown) and hooking them up to the model, shown in Listing 8.

```
- changedF:sender {
    self f:sender intValue.
}
- changedC:sender {
    self c:sender intValue.
}
- changedK:sender {
    self k:sender intValue.
}
- f:degreesF {
    self basicC:(degreesF - 32) / 1.8.
    ivar:ui/celsiusTextField/intValue := self c.
    ivar:ui/kelvinTextField/intValue := self k.
}
- f {
    ^ self c * 1.8 + 32.
}
- k:degreesK {
    self basicC:(degreesF - 273.15.
    ivar:ui/celsiusTextField/intValue := self c.
    ivar:ui/fahrenheitTextField/intValue := self f.
}
- k {
    ^ self c + 273.15.
}
- c:newValue {
    self basicC:newValue.
    ivar:ui/fahrenheitTextField/intValue := self f.
    ivar:ui/kelvinTextField/intValue := self k.
}
- c {
    ^ivar:c.
}
- basicC:newValue {
    defaults:c := ivar:c.
    ivar:c := newValue.
}
```

Listing 8. Adding Kelvin scale

Most of the code consists of straightforward if tedious additions, except for the code keeping the UI in sync with the model. Every method that sets a new value for a specific temperature has to be modified to update the respective other temperature text fields (assuming that the change originated in the UI).

At this point, it becomes clear that our original strategy of updating the UI from the individual setter methods is probably not tenable in the long run. Listing 9 replaces this distributed logic with a centralized `-updateUI` method that updates all of the UI from the model. This method is only invoked from the `-c:` accessor method. While this change simplifies the code, it removes the optimization that prevented updating the UI element that initiated the change.

The solution in Listing 9 starts to approximate a true Model View Controller (MVC) [27][29] approach. However, the UI elements are widgets, not classical Views, so they contain their own data rather than referring to and refreshing themselves from the model. Updated data must therefore be pushed to them, they cannot pull it after receiving a `#changed` notification. Actually implementing the MVC pattern in this instance would therefore entail introducing an intermediate layer that mediates between the widgets and the model, listening to `#changed` notifications and pulling data from the model and pushing to the widgets.

2.6 Discussion

As we have seen, even a conceptually very simple application such as a temperature converter quickly attracts significant complexity with non-obvious trade-offs once the requirements of an interactive version of that application are taken into account.

This complexity is not the result of a complicated domain model, but rather of the architectural embellishments required to move data from location to location in order to keep the different

```
-updateUI {
    ivar:ui/celsiusTextField/intValue := self c.
    ivar:ui/fahrenheitTextField/intValue := self f.
    ivar:ui/kelvinTextField/intValue := self k.
}
- f:degreesF {
    self c:(degreesF - 32) / 1.8.
}
- f {
    ^ self c * 1.8 + 32.
}
- k:degreesK {
    self c:(degreesF - 273.15.
}
- k {
    ^ self c + 273.15.
}
- c:newValue {
    ivar:c := newValue.
    self updateUI.
}
- c {
    ^ivar:c.
}
```

Listing 9. Centralized UI update

parts of the application (model, user interface, persistence) synchronized. In the next section, we will look at a mechanism for simplifying this kind of overhead.

3. Constraint Connectors

Constraint connectors are a mechanism for addressing the architectural problems faced by interactive applications. They generalize the concept of a one-way dataflow constraint [47] from an element that is part of a constraint solver to a general architectural element that can be implemented in many different ways, including using a constraint solver.

A constraints connector connects at least two *constraint variables*, one *target variable* and one or more *source variables*. Constraint variables are storage locations that have a mechanism for detecting that they have been changed and somehow notifying the rest of the system of such a change. The relationship between these variables is defined by using a mix of declarative and procedural techniques. The fact that there is a relationship that should hold between the variables is declared by their participation in the constraint. What the exact nature of the relationship is is defined procedurally using an *update method* that computes the value of the target variable from the source variables.

When a source variable changes, the constraint is no longer satisfied, the update method must be run to re-satisfy the constraint.

3.1 Push Evaluation

The simplest way of implementing an equality constraint is when the source variable knows about the target variable and new values can simply be pushed to the target whenever the source changes.

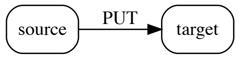

Figure 1. Push Evaluation

Architecturally, there must be at least a direct variable reference to the target that is available to the source and offering at least a PUT operation, or more generally an update procedure. See Figure 1.

With Polymorphic Identifiers, this reference can be any type of resource, be it an in-memory variable, file or network resource.

3.2 Pull Evaluation

When the target knows about the source, but not vice-versa, push evaluation is not possible. This can be due to physical constraints, for example a server in a client-server may not have a permanent connection to its clients, and therefore cannot inform the clients of changes to data. It can also be due to architectural constraints, typically to avoid cyclical dependencies.

In the case of pull evaluation (see Figure 2), the architecture of the constraint connector must contain at least a variable reference from target to source and a GET operation or update procedure. In addition, t here must also be some mechanism for communicating to the target that the source is out of date.

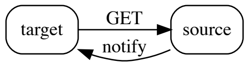

Figure 2. Pull Evaluation

Most typically this update/out-of-date notification is some form of implicit invocation mechanism [37], but this is not necessary. All that is needed is some way of notifying the target that the source has changed and therefore the target is out of date and must be updated. If all else fails, the target can regularly poll the source to check for changes. Although this is not a typical (and likely inefficient) implementation of the notification mechanism, it is sufficient for a pull-evaluation dataflow constraint.

3.3 Constraint Solvers

Dataflow constraint solvers such as Deltablue [23], ThingLab [11] or Amulet/Garnet [47] are a convenient way of implementing constraint connectors. Internally, they use what are essentially constraint connectors with either push or pull evaluation. In addition, they often maintain global knowledge of the entire constraint graph, which can make evaluation of updates more efficient.

However, it is not necessary to implement all (or even any) parts of an application using a constraint solver in order to use constraint connectors. All that is needed are code components that implement the communication patterns matching the general structure of either a push- or pull-evaluated constraint connector.

3.4 Model View Controller

Model View Controller [29] is a widespread pattern for keeping one or more views synchronized with a model and vice versa. The model ⇒ view update pattern (see Figure 3) closely matches the pull evaluation style constraint connector. With constraint connectors as a language feature, we can directly express the view update part of the MVC pattern in code, regardless of how the pattern is actually implemented.

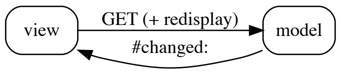

Figure 3. MVC View Update

The controller ⇒ model communication on the other hand largely matches a push evaluation constraint connector, at least if the views in questions are widgets that maintain their own internal state.

3.5 Filesystem

The clear definition of the pieces of a constraint connector makes the implementation pluggable. With a `kqueue` [28] based file change notification mechanism, it becomes possible to integrate file maintenance into the system, for example if communication is file-based or to implement `make`-style file maintenance rules such as that shown in Figure 4.

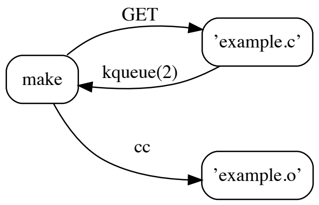

Figure 4. make

4. Evaluation

We can now rebuild the temperature converter from Section 2 using constraint connectors. We use a variant of the DeltaBlue [23] incremental dataflow constraint solver embedded into Objective-Smalltalk and made applicable to multiple domains by making variable access and change notification mechanism flexible using Polymorphic Identifiers.

Syntactically, one-way dataflow constraints are expressed using the `|=` connector. The constraint expression `lhs |= rhs` is very similar to the assignment expression `lhs := rhs`, the difference being that an assignment is evaluated once, whereas the constraint is maintained by the system, a sort of perpetual assignment. The variable on the left hand side is constrained to be equal to the formula (update procedure) on the right hand side. Whenever a variable referenced on the right hand side is changed, the left hand side is updated by re-evaluating the update formula.

DeltaBlue handles multi-way constraints, so multiple update formulae referencing the same variables are combined into a single constraint. In the case of the constraints being simple equality, rather the complex formulae, the bi-directional constraint connector `=|=` can be used as a shorthand for specifying the two symmetric update formulae `a |= b` and `b |= a`.

The solver handles the cycles created with multi-way constraints by only considering one direction of a multi-way constraint in an update cycle.

The initial model shown in Listing 10 has mostly superficial differences from the initial example in Listing 3: instead of a single variable for Celsius, both Fahrenheit and Celsius are represented as instance variables. The constraints specifying their relationships are encoded directly and independently from setters and getters, which are hidden.

```
ivar:f |= (9.0/5.0) * ivar:c + 32 .
ivar:c |= (ivar:f - 32) * (5.0/9.0).
```

Listing 10. Basic Temperature Converter Model

The update logic is triggered automatically whenever a variable is modified to keep the other variable in sync.

4.1 Adding User Interface

The architectural differences become more noticeable when adding UI, as shown in Listing 11. The code to hook up the UI is purely additive, which is why Listing 11 only shows the additions. It defines two additional bi-directional dataflow constraints that keep the instance variables synchronized with their respective UI text fields.

```
ivar:ui/celsiusTextField/intValue    =|= ivar:c.
ivar:ui/fahrenheitTextField/intValue =|= ivar:f.
```

Listing 11. Adding UI

The original model code does not need to be updated, because the update logic is implicit in the constraint definitions. Optimally updating only the values that have changed, so for example only the dependent values when changing is also implicit in the connector definition, and implemented behind the scenes in the constraint solver.

4.2 Adding Persistence

Adding persistence is as easy as adding a constraint from one of the temperature instance variables to the persistent variable, it doesn't really matter which.

```
ivar:c := defaults:celsius.
defaults:celsius |= ivar:c.
```

Listing 12. Adding Persistence

Referring to the persistent variable using the Polymorphic Identifier `defaults:celsius` makes it possible to place it on the left hand side of a constraint connector (`|=`), something that would have been much more difficult with the method API (Listsing 6, lines 3-4).

4.3 Adding a Temperature Scale

This time, adding the Kelvin temperature scale is also purely additive. We add an instance variable `ivar:k` to hold the temperature in degrees Kelvin (not shown), connect an additional text field to that field and define additional constraints relating the Kelvin and Celsius instance variables. These additions are shown in Listing 13.

```
ivar:ui/kelvinTextField/intValue     =|= ivar:k.
ivar:k |= ivar:c + 273.15.
ivar:c |= ivar:k - 273.15.
```

Listing 13. Adding a Temperature Scale

This brings us to the (almost) complete temperature converter application shown in Listing 14. It shows all the elements of the application and how they interact. It is not only more compact than the non-constraint version in Listing 9, but actually handles a few details that were elided in that version, such as initialization from persistence and hooking up the text fields to the model.

The complete program is the simple concatenation of the individual pieces, no modifications of previous code was necessary.

4.4 Deriving High Level Architecture

Grouping the three text fields into the `ui`, the three temperature variables into `memory-model` and the defaults database access into `persistence`, we arrive at the high-level architecture shown in Listing 15.

Unlike the previous listings in this section, Listing 15 is pseudo code, because we don't yet have the required grouping mechanism for constraints. However, the distance from this higher-level

```
ivar:ui/celsiusTextField/intValue    =|= ivar:c.
ivar:ui/fahrenheitTextField/intValue =|= ivar:f.
ivar:ui/kelvinTextField/intValue     =|= ivar:k.

ivar:f |= (9.0/5.0) * ivar:c + 32 .
ivar:c |= (ivar:f - 32) * (5.0/9.0).
ivar:k |= ivar:c + 293.
ivar:c |= ivar:k - 293.

ivar:c := defaults:celsius.
defaults:celsius |= ivar:c.
```

Listing 14. Complete Temperature Converter

```
ui           =|= memory-model.
memory-model  := persistence.
persistence   |= memory-model.
```

Listing 15. High Level Architecture of Temperature Converter

grouped description to the lower-level ungrouped implementation in Listing 14 is small and straight-forward to bridge.

4.5 Discussion

The overall architecture of the temperature converter shown in Listing 15 is also almost identical to the task management client application architecture shown in Listing 1, which is as it should be because both applications have a UI that interacts with an in-memory model that is persisted. The main difference is that the task management application additionally synchronizes via a cloud-based backend, and that difference is readily apparent from comparing the two listings.

Another result of the similarity between these two listings is that the constraint-connector formulation of the architecture appears to clearly separate functional/domain aspects from architectural/structural aspects.

5. Application

Having constructed a toy example application with constraint connectors provided by a constraint solver bottom up, we are now going to deconstruct the architecture of a real world task management application using constraint connectors top down.

The use of constraint connectors was not a planned characteristic of the system, but rather something that emerged as the application was built: different parts of the system needed to be kept in sync which each other both within the process and across process and machine boundaries. Once the mechanisms had been built, it became apparent that in each case the individual pieces that had been built were parts of a constraint connector.

As the application is built with Objective-C, there is also no direct linguistic support for those constraint connectors, instead they are present both in a non-executable architecture description and conceptually in the code.

The architecture in question (see Listing 1) is shown again graphically in Figure 5, with the solid arrows representing dataflow constraints and the open arrow representing a one-time assignment. The fact that this diagram is isomorphic to the textual representation in Listing 1 opens up the intriguing possibility of having a meaningful boxes-and-arrows graphical representation [43] of the overall architecture of interactive systems.

This architecture shows the large scale structure of the application. It is not complete, but can be refined to be complete. The reminder of the section will demonstrate this refinement from overall architecture to implementable components using the four top-level components and three top-level connections of the application.

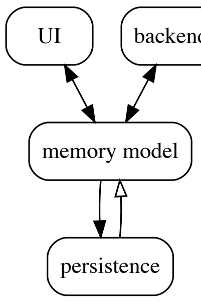

Figure 5. Overall Wunderlist Architecture

5.1 Memory-Model and Object References

The model component is implemented as plain Objective-C objects, for example `WLTask` representing a task and `WLList` representing a list. These objects are stored and served from an in-process REST server [51] implemented as nested dictionaries.

Objects are referred to using `ObjectReference` instance, which correspond to URIs in the REST model. An object reference (see Listing 16) defines an access path to a specific object or set of objects: `wlstore:<entity>/container/<id>/object/<id>`. Every model object can also create an `ObjectReference` pointing to itself.

Object references are structured and can be transformed. Removing the object id yields a new object reference that refers to all the objects in the same containing object as the original object, for example the list a task was in.

```
@interface WLObjectReference : NSObject

@property (nonatomic, assign) wlid objectID;
@property (nonatomic, assign) wlid containerID;
@property (nonatomic, strong) NSString *entityType;

- (BOOL)isMatchedByReference:(WLObjectReference *)ref;
- (BOOL)matchesObject:(WLObject *)anObject;

@end
```

Listing 16. WLObjectReference interface

Having an extensional representation of object references rather than just object pointers makes it possible to refer to objects without having the actual object at hand, for example because it doesn't exist yet, hasn't been fetched yet or has been deleted. These object references are also trivially serializable and can be used to derive both disk addresses and web URIs, again without requiring an actual object to be present.

With object references, every part of the system can easily implement the GET part of the *pull* style constraint connector by locally storing an object reference and issuing a GET to the in-process REST server with that reference whenever necessary.

5.2 Notifications

The *notification* part of the constraint connector is implemented throughout the system using a `WLRESTOperationQueue`. Each `WLRESTOperation` consists of an object-reference (i.e. a URI) and an operation (GET/PUT/POST/DELETE). It does not contain an actual model object, because the purpose is only to notify clients that there has been a change to the memory model and take appropriate action.

The application currently has three such instances of the class `WLRESTOperationQueue`: one connecting the memory-model to the UI, one connecting the memory-model to the persistent store and finally one connecting the memory-model to the network interface.

In each case the purpose is the same: the memory-model has changed, please update the user interface, on-disk representation or back-end with the new value. In each case, the target subsystem then executes a subsystem-specific update method to effect that change.

Notifications are buffered in order for the individual subsystems to run on different threads and to account for the different speeds of the subsystems. Only passing references and not the actual data means that the target subsystem will always retrieve the most current data from the in-process REST server, not necessarily the data at the time the original notification was enqueued. This allows the notification queues to prune duplicates, lessening the load on the target subsystems.

5.3 Persistence

The simplest subsystem attached via a constraint connector is the persistence component. It stores objects on disk as serialized JSON files, with objects grouped into buckets defined by their parent object (the container in the object reference). As an example, tasks are stored in buckets defined by their list, subtasks or comments in buckets defined by their enclosing task.

An independent actor is responsible for persisting objects to disk. It is connected to the memory model via a one-way constraint connector, shown in Figure 6. Whenever the memory model is modified, it posts the object reference of the modified object to this `WLRESTOperationQueue`.

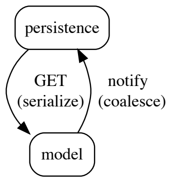

Figure 6. Model ⇒ Persistence constraint

The queue is configured to remove the object-id from its incoming object references. This means the reference now no longer points to, for example, a specific task, but rather the entire set of tasks that belong to a list (but not the list object itself). Combined with the uniquing performed by the queues, this leads to automatic write-coalescing, both to combine writes to the same bucket and to only write the most current version of an object that has multiple changes.

The actor executes the update procedure by reading the object reference, which now refers to a bucket, from the queue, fetching a bucket's worth of objects from the in-process REST server, serializing the objects to JSON and writing the result to a file whose name is derived from the object reference. At the end of the update procedure, the dataflow constraint is satisfied again: the on-disk representation matches the memory model.

The memory-model does not need to know about the persistence mechanism, apart from publishing "changed" notifications to its notification queue.

Reading from disk is done independently and lazily at startup, so there is no constraint connector from persistence to the memory model. We do not expect outside processes to alter the persistent store.

5.4 User Interface

The constraints between user interface component and model (shown in Figure 8) are more complex than for the persistence component. First, the user typically also wants to be able to make modifications, so in addition to the pull-based constraint going from model to ui, there is a push-based constraint going from ui to model. In both cases, only the UI has a direct reference to the model, the model once again only publishes "changed" notifications.

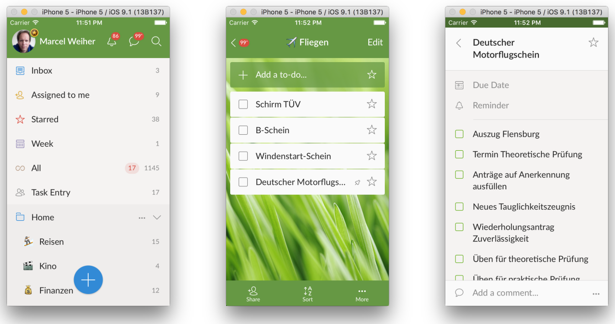

Figure 7. Principal Wunderlist UI Elements

The Second complication is that the user interface component is a composite, consisting of many individual subcomponents. The full detail of the user interface is beyond the scope of this paper, but the major subcomponents are the following, shown left to right in Figure 7: the *sidebar* that shows all the task lists, a *list view* that shows all tasks in a single list and finally a *detail view* that allows viewing and manipulation of a single task. The views are related among themselves via navigation/selection actions and to the model via the same constraint connector indicated in the overall model.

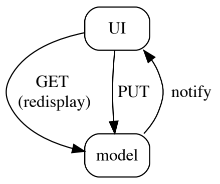

Figure 8. Model ⇔ User Interface Constraints

The connections shown in the overview (Figure 8) distribute over these components. With the addition of the selection/navigation actions connecting the views, this leads to the implementation-oriented diagram in Figure 9

Figure 9 is very detailed and difficult to parse, because the constraint connectors are depicted as their constituent connectors, which are always the same (parametrized by the specific object reference). Figure 10 simplifies the connection diagram by showing the bi-directional constraint connectors as single double-headed lines. Using the same symbol is justified by the fact that these are, in fact, always variations of the same kinds of connector.

Similar to the connection to the persistence component, coalescing is also used when notifying the UI. As with disk updates,

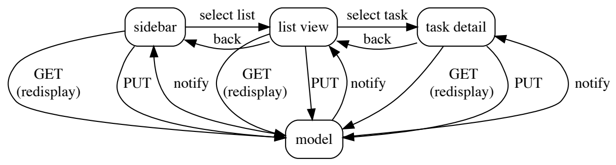

Figure 9. UI Component Details

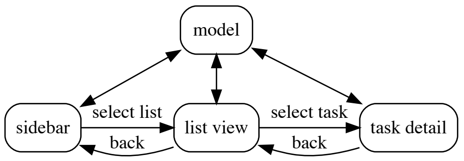

Figure 10. UI Components with Constraint Connectors

it makes no sense to update the UI either incorrectly with information that is no longer current or redundantly with information that was already displayed. In addition, both the UI itself and the human reading the UI have an ability to display/absorb changing information at a rate that is much lower than the rest of the system can generate, especially with the heavy use of animations in today's interfaces, which simply require a minimum time to display. On the other hand, the requirements for update latency are very stringent, so simply batching and delaying updates is also not acceptable.

Adapting the update coalescing mechanism described in the previous section and making it dynamic solves the problem of combining the requirements of low-latency and high-fidelity with the ability to throttle down in order not to overwhelm the user interface in high load situations: in low-rate situations, every store reference is passed through the update queue unchanged. Every time an update notification is delivered to the UI, notification delivery is halted for a small amount of time in order to give the UI a chance to update, the user a chance to absorb the information and animations the chance to finish.

During the time that delivery is disabled any further UI update notifications accumulate in the notification queue, so the first update is always immediate and delays only occur when the UI is already busy. If the rate of change exceeds the ability of the UI to absorb the changes, the coalescing mechanism used in the persistence connection is activated dynamically, with more and more detail being removed from each object reference. With the reference uniquing this will for example mean that update notifications for several individual tasks in a list will be coalesced into a single update for the entire list. If even that is insufficient to allow the UI to catch up, coalescing is increased further until finally what remains is only a blanket notification that something has changed and the entire UI needs to be refreshed.

Being able to use the same connector with only slight variations in two seemingly very different situations came as a surprise to the development team.

5.5 Network Communication

The network interface once again uses the same connectors, and once again in slightly different variations and combinations. Internally, the same type of `WLRESTOperationQueue` is used to notify the network interface of changes to the model that need to be communicated to the backend.

More interestingly though, the actual network connection to the backend is also a constraint connector, though this time one implemented with HTTP and a WebSocket connection as shown in Figure 11. The push/pull direction is reversed in this connection, it is the model that is informed of changes in the backend via a WebSocket connection and then uses an HTTP GET operation to fetch the updates. The push constraint is for sending updates to the server.

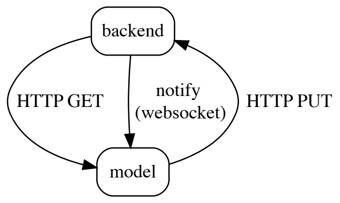

Figure 11. HTTP/Websocket based constraint connector

If the WebSocket fails, the notification part of the constraint connector is replaced by polling via HTTP.

Another difference that is beyond the scope of this paper is that the GET part of the constraint connector is asynchronous in this case, due to the high and unpredictable latencies of network operations.

This is by far the most complex constraint connector, with the implementation trying very hard to optimize both bandwidth and latency, yet in terms of interface it looks identically to the other constraint connectors.

5.6 Discussion

In building Wunderlist, we discovered that variations of constraint connectors turned out to be fundamental application building blocks, not just conceptually and architecturally, but also in terms of the reusable capabilities they provided to the application.

One lesson was the importance of being able to use different types of constraints (for example push evaluation vs. pull evaluation) as needed, as well as being able to vary the implementation of parts of the constraint connectors, particularly the notification part.

Another important enabler was the use of Polymorphic Identifiers as flexible, non-pointer based object references. These object references made possible both the in-process REST server that serves as the source of the GET part of the pull-evaluated constraints as well as the notification queues that complete the constraint connectors.

These mechanisms by themselves have been tremendously helpful in building a high quality, responsive application and keeping the code simple, factored and compact. The realization that these component parts make up constraint connectors has opened up many more avenues for simplification and streamlining.

6. Related Work and Future Directions

Although there has been a lot of work on using programming to solve constraint problems, there hasn't been nearly as much on using constraints to solve programming problems. For example, the Module System for Constraints in Smalltalk [21] does not use constraints to enhance Smalltalk modularity, but instead creates a module system to make constraint evaluation more efficient.

We view constraints as structured programming for variables, making it possible to capture and declaratively specify higher level interactions with or between variables than just load and store.

6.1 Software Architecture and Modeling

Various publications on software architecture, including the seminal *Software Architecture, Patterns of an Emerging Discipline* [44], *Taxonomy of Software Connectors* [30], and *Software Architecture: Foundations, Theory, and Practive* [46] contain what are considered to be comprehensive lists of connector types. None list a constraint connector, though they do list the constituent connectors (data access, notification).

Clafer (class, feature, reference) [6] is a modeling language that allows specifcation of both architectural meta-models and domain-oriented feature-models as well as their combinations. As discussed in section 4.5, Listing 1 represents a meta-model of a certain class of interactive applications, independent of their specific features, with further refinement then adding the domain-specific features. Transporting Clafer's approach to integrating meta- and feature-models from the modeling level to Objetive-Smalltalk's implementation level seems like a promising approach for connecting the high-level architectural descriptions to constraint-based or non-constraint-based implementations. Clafer allows constraints on the implementation to be specified and then checked via constraint solvers such as Z3 [15] or Alloy [26]. Having such similar mechanisms at different levels of abstraction is intriguing and possibly opens up meta-circular definitions and implementations, further reducing the gap between models and implementations.

Task models describe the logical activities that have to be carried out in order to reach the users goals and allow user interfaces to be automatically generated from those models [8]. They are complementary to constraint connectors in several ways: constraint connectors are mostly about the structure of implementations, whereas task models are more about modeling user features, with implementations being (automatically) generated from those models. Furthermore, task models describe goals that typically involve changes to the system state or requests for information, whereas constraint connectors allow the system to return to a steady states by enforcing invariants.

Bergman et al. [7] describe a system where task models are used for specifying an application generically to run on different devices and constraints used in the specification of those user interfaces. Although constraint connectors are not primarily intended for layout constraints, they could easily be extended in that direction (see next section).

ArchJava [1] is an extension to Java that claims to connect architecture to implementation. While extending Java with concepts such as components, connectors and ports and even allowing custom connectors to be defined, ArchJava limits connectors to be method-based, with ports specifying `required` and `provided` methods in a fashion that is not much different from interfaces declared by classes or required of instance variables. The addition of `broadcast` methods on ports helps with the implementation of the notification part of pull-based constraint connectors. Full-fleged constraint connectors, however, connect via their constraint variables and requires composition of primitive connectors, both of which ArchJava cannot express.

6.2 Generalized Constraint Solving

Babelsberg [18] integrates a variety of solver libraries such as Z3 [15] and Cassowary [5] with different existing programming languages such as JavaScript, Ruby and Smalltalk. It allows the the user to specify constraint problems directly in the host programming language and then have those problems solved by the libraries. The comprehensive integration between constraints and host imperative languages coupled with powerful solvers led to sometimes surprising results, which required placing strict limits on the solvers and the integration in order to make results more predictable [19].

Constraint connectors don't try to find solutions to potentially under- or overconstrained computational problems, but focus on the more tractable problem of maintaining system invariants using simpler dataflow constraints, making them more suitable as a general purpose programming mechanism. It would be interesting to explore unifiying the mechanisms in order to have more complex constraint solvers available where suitable, for example in order to validate model constraints.

Kaleidoscope [22] is a language with direct support for constraints. It features multiple object views and constraint constructors for breaking user-defined constraints down into primitive constraints supported by the built-in solver. In Kaleidoscope, variables determined by constraints must be slots of objects and defined using the solver. The use of Polymorphic Identifiers allow constraint connectors to define arbitrary resources, including external files or web resources, and constraint implementations are not limited to the abilities of a specific solver. It is unlikely that Kaleidoscope's constraints would have been expressive enough to implement the network-based connector in Section 5.5.

6.3 Constraints in Graphics and UI

Whereas software architecture focuses on the communication between and connection of components, constraint programming has been concerned primarily with solving computational problems such as graphical layout, starting with what is said to be the very first constraint solver, Sutherland's 1957 Sketchpad [45] system, which allowed the user to specify constraints between graphical objects.

As an example, a square was defined as four lines that are constrained to be the same length, perpendicular to each other with certain shared points. Sketchpad then used a relaxation based solver to come up with the square solution to these constraints. Although very powerful,it is unclear whether this constraint-based description of a square is more useful than a simple procedural one.

Almost four decades after Sketchlab, the paper describing the Cassowary linear constraint solver for user interface applications [5] used constraints to demonstrate Varignon's theorem: connecting the midpoints of any quadrilateral forms a parallelogram. Again, the power of a multi-way solver for linear constraints seemed overkill, a simple `InsetQuadrilateral` object could easily compute the required points and their connecting lines procedurally without resolving to complex constraint solving methods.

These and other examples led to the conclusion that for many examples of constraints, the usefulness lay not so much in their computational abilities, but rather in the fact that they could create new abstractions by describing relationships between existing abstractions without having to modify those abstractions.

Experience reports about the work on the Garnet and Amulet constraint toolkits [47] were extremely helpful in shaping constraint connectors. The influences are too many to list completely, one example is the difficulty with both DSLs (too restricted) and general purpose languages (cumbersome syntax for constraints), which led to our approach of co-evolving a general purpose language with the constraint connector mechanism in order to achieve good semantic and syntactic integration (almost DSL-like) without sacrificing generality.

The authors also report the relative unimportance of the specific constraint solving algorithm, which ultimately led to another central feature of the constraint connector mechanism: allowing diverse constraint update mechanisms, which then led to the applicability of the technique to non-layout tasks, which the authors found difficult.

Apple incorporated the Cassowary linear constraint solver as the standard view layout mechanism into both Mac OS X and iOS under the name of *Autolayout* [2]. In addition, OS X supports a simple dataflow constraint binding mechanism under the name *Cocoa Bindings* [4].

Both systems suffered from problems of missing linguistic integration, leading to verbose and unwieldy APIs that have to specify equations as method arguments with enums or strings. Listing 17 is the equivalent of the following simple arithmetic constraint: `button.width |= 0.5 * superview.width + 3.0`.

```
NSLayoutConstraint constraintWithItem:button
                     attribute:NSLayoutAttributeWidth
                     relatedBy:NSLayoutRelationEqual
                     toItem:superview
                     attribute:NSLayoutAttributeWidth
                     multiplier:0.5  constant:3.0.
```

Listing 17. Specifying an Autolayout constraint in code

In addition to syntactic issues, Autolayout also turned out difficult to use due to the usual problems of over- and under-constrained constraint systems leading to surprising solutions, so much that Apple later had to introduce a simpler box-layout mechanism on top.

Continuing the theme of integrating results from research and industry and potentially building on further work integrating more complex solvers (Section 6.2), generalizing constraint connectors to interface with and/or supplant Autolayout and Cocoa Bindings could be a very interesting avenue for future work.

(If bi-directionality were desired, the `=|=` connector could also be used for the layout constraint, but see Section 6.5 below). Another useful addition would be to expand on the work on visual constraint debugging [49] and apply it too all of these areas.

Constraints as a Design Pattern [41] addresses the predictability and integration issues by treating constraints as programming elements that can be flexibly integrated with regular programming. Like constraint connectors, the top-level architecture of applications is described using constraints and these can then be implemented in different ways. However, it still focuses on computation (components) rather than architectural connection, and on the algorithmic problems of user interface programming rather than the problems caused by architectural mismatch.

Interstate [38] defines a visual and tabular language with constraints and state machines specifically for user interface programming. It uses a fixed constraints solver that neither permits the variation in notification mechanisms we found useful in Wunderlist, nor is it capable of dealing with non-object resource types.

6.4 Spreadsheets Constraints

The one-way dataflow constraints used in Garnet and Amulet are also referred to as "spreadsheet constraints", because the way they automatically update all dependent calculations is effectively the same as an electronic spreadsheet [48] [47]. Although constraint connectors can use multiway constraints, these are used in a stylized fashion that reduces to sets of one-way constraints with cycle detection.

The success of spreadsheets as an end user programming technology where so many others have failed is attributed in part to the computational model that shields the user from many low-level details including control flow, the immediate visual feedback in the two-dimensional grid and the motivational factor from achieving real results in a few hours [35].

The fact that many if not most of the applications of spreadsheets are not for graphical layout of user interface layout also seems to contradict the findings by vander Zanden on the scope for the use of spreadsheet constraints being limited to UI layout.

The attraction of combining the direct appeal, productivity and end-user success of spreadsheets with general purpose programming (languages) is obvious, and there have been many attempts to combine the two. Among these the Analyst Spreadsheet [40] allowed arbitrary Smalltalk expressions in every spreadsheet cell (identifiers extended with the ability to locate cells) and in turn allowed arbitrary Smalltalk objects to be displayed as a result, whereas the dataflow language Lucid was extended to create a 3D intensional spreadsheet [17].

An interesting research question would be whether having a spreadsheet-like constraint mechanism that removes a lot of low-level details (see Section 4) not just available as a built-in language mechanism (rather than as a bolt-on), but also widely used in the system (see the previous section), could significantly reduce the semantic gap and make end-user programming more feasible. A related question is whether having both constraints and polymorphic identifiers turns a spreadsheet from a complex application to a simple library, and whether that could also lower the semantic gap compared to having either special-purpose formula languages or general purpose formula languages that do not intrinsically support spreadsheet constraints.

6.5 Bidirectional Programming

In section 4, we used plain imperative expressions as the update procedures, forcing us to specify two uni-directional update-procedures `|=` for the temperature conversion, whereas simple equality constraints could be specified using the bi-directional constraint connector `=|=`. Thinglab [12] instead composed more complex formulae from primitive bi-directional constraints such as multiplication (with a constant) and addition (of a constant), which automatically resulted in a bi-directional constraint with only one formula specified, but limits expressible constraints to ones that can be constructed with the pre-defined library. Cassowary restricts constraints to linear equations and inequalities, which are naturally bi-directional.

Sticking with advice from the Garnet/Amulet team, we decided to use update procedures having the full power of the imperative base language in order to preserve general applicability, which precludes automatic bi-directionality because not all update procedures can be inverted

Since we are co-evolving the language with the constraint system, we do have the option of revising this decision in the future, and the `=|=` syntax makes it possible for users to demand bi-directional evaluation. The system can then attempt to find or construct an inverse, or select a solver that can accept bi-directional constraints (such as Cassowary) and verify that the formula is sufficiently limited for the solver to be applicable (linear equation or inequality). If no inverse or applicable solver for the formula can be found, the system signals a compile error.

Lenses provide an interesting general approach for expanding the range of possible bi-directional expressions to tree transformations [20], relational algebra [9] and strings [10], with *lens combinators* to construct larger bi-directional transforms from primitive ones. Basic lenses only deal with (bi-directional) update procedures, Parametric lenses [16] add notifications in order to make the system structurally very similar to dataflow constraints.

Lenses were billed as a possible solution the the *View Update Problem* from database theory, which is the problem of reflecting updates made on a view of a database correctly back to the underlying database. "In summary, our theorems show that there are very few situations in which view updates are possible–even fewer, in fact, than intuition might suggest." [14].

7. Summary and Outlook

We have demonstrated selected problems caused by the architectural mismatch between current programming languages and the architecture of interactive applications. We showed that using dataflow constraint connectors can elegantly solve some of these problems, and that direct linguistic support for these connectors is very helpful.

Directly expressing these constraint connectors in code has the potential for bringing architecture and implementation for certain systems much closer than they are today, as well as bringing niche applications of constraints such as UI layout or build systems into the mainstream of programming. In addition, end-user programming might benefit from reducing the semantic gap between successful technologies such as spreadsheets and the difficulties of general purpose scripting and programming.

However, the polymorphic nature of these connectors means that direct linguistic support, i.e. built in support for some specific type of constraint is not sufficient. Instead, we need to take one step up the abstraction ladder and provide support for adding user-defined connectors and turning them into first class abstraction/decomposition mechanisms on par with built-in mechanisms such as procedure calls or message sends.

In fact, just like pure object-oriented languages discarded the notion of built-in concrete data types except as an implementation expedience, a pure architectural language would probably discard the notion of built-in connectors, conceptually treating all connector types (procedure call, assignment, message send, constraint, etc.) equally

References

[1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 187–197, New York, NY, USA, 2002. ACM. ISBN 1-58113-472-X. . URL http://doi.acm.org/10.1145/581339.581365.

[2] Apple Inc. Auto Layout Guide. URL https://developer.apple.com/library/mac/documentation/UserExperience/Conceptual/AutolayoutPG/index.html.

[3] Apple Inc. Cocoa Reference, 2009. URL http://developer.apple.com/docs/Cocoa/.

[4] Apple Inc. What Are Cocoa Bindings?, 2015. URL https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CocoaBindings/Concepts/WhatAreBindings.html.

[5] G. J. Badros, A. Borning, and P. J. Stuckey. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, Dec. 2001. ISSN 1073-0516. . URL http://doi.acm.org/10.1145/504704.504705.

[6] K. Bak, K. Czarnecki, and A. Wasowski. Feature and Meta-models in Clafer: Mixed, Specialized, and Coupled. In *Proceedings of the Third International Conference on Software Language Engineering*, SLE'10, pages 102–122, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19439-9. URL http://dl.acm.org/citation.cfm?id=1964571.1964581.

[7] L. D. Bergman, T. Kichkaylo, G. Banavar, and J. B. Sussman. Pervasive Application Development and the WYSIWYG Pitfall. In *Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction*, EHCI '01, pages 157–172, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-43044-X. URL http://dl.acm.org/citation.cfm?id=645350.650726.

[8] L. Birnbaum, R. Bareiss, T. Hinrichs, and C. Johnson. Interface Design Based on Standardized Task Models. In *Proceedings of the 3rd International Conference on Intelligent User Interfaces*, IUI '98, pages 65–72, New York, NY, USA, 1998. ACM. ISBN 0-89791-955-6. . URL http://doi.acm.org/10.1145/268389.268400.

[9] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational Lenses: A Language for Updatable Views. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 338–347, New York, NY, USA, 2006. ACM. ISBN 1-59593-318-2. . URL http://doi.acm.org/10.1145/1142351.1142399.

[10] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful Lenses for String Data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 407–419, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. . URL http://doi.acm.org/10.1145/1328438.1328487.

[11] A. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Trans. Program. Lang. Syst.*, 3:353–387, October 1981. ISSN 0164-0925. . URL http://doi.acm.org/10.1145/357146.357147.

[12] A. H. Borning. *Thinglab–a Constraint-oriented Simulation Laboratory.* PhD thesis, Stanford, CA, USA, 1979. AAI7917213.

[13] S. Chatty. Programs = Data + Algorithms + Architecture: Consequences for Interactive Software Engineering. In J. Gulliksen, M. B. Harning, P. Palanque, G. C. Veer, and J. Wesson, editors, *Engineering Interactive Systems*, pages 356–373. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-92697-9. . URL http://dx.doi.org/10.1007/978-3-540-92698-6_22.

[14] U. Dayal and P. A. Bernstein. On the Updatability of Relational Views. In *Proceedings of the Fourth International Conference on Very Large Data Bases - Volume 4*, VLDB '78, pages 368–377. VLDB Endowment, 1978. URL http://dl.acm.org/citation.cfm?id=1286643.1286692.

[15] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL http://dl.acm.org/citation.cfm?id=1792734.1792766.

[16] L. Domoszlai, B. Lijnse, and R. Plasmeijer. Parametric Lenses: Change Notification for Bidirectional Lenses. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages*, IFL '14, pages 9:1–9:11, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3284-2. . URL http://doi.acm.org/10.1145/2746325.2746333.

[17] W. Du and W. W. Wadge. A 3D Spreadsheet Based on Intensional Logic. *IEEE Softw.*, 7(3):78–89, May 1990. ISSN 0740-7459. . URL http://dx.doi.org/10.1109/52.55232.

[18] T. Felgentreff, A. Borning, R. Hirschfeld, J. Lincke, Y. Ohshima, B. Freudenberg, and R. Krahn. Babelsberg/JS. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 411–436. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44201-2. . URL http://dx.doi.org/10.1007/978-3-662-44202-9_17.

[19] T. Felgentreff, T. Millstein, A. Borning, and R. Hirschfeld. Checks and Balances: Constraint Solving Without Surprises in Object-constraint Programming Languages. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 767–782, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. . URL http://doi.acm.org/10.1145/2814270.2814311.

[20] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for Bi-directional Tree Transformations: A Linguistic Approach to the View Update Problem. *SIGPLAN Not.*, 40(1):233–246, Jan. 2005. ISSN 0362-1340. . URL http://doi.acm.org/10.1145/1047659.1040325.

[21] B. N. Freeman-Benson. A Module Mechanism for Constraints in Smalltalk. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 389–396, New York, NY, USA, 1989. ACM. ISBN 0-89791-333-7. . URL http://doi.acm.org/10.1145/74877.74918.

[22] B. N. Freeman-Benson. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. *SIGPLAN Not.*, 25(10):77–88, Sept. 1990. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/97946.97957`.

[23] B. N. Freeman-Benson, J. Maloney, and A. Borning. An Incremental Constraint Solver. *Commun. ACM*, 33(1):54–63, Jan. 1990. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/76372.77531`.

[24] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Softw.*, 12(6):17–26, Nov. 1995. ISSN 0740-7459. . URL `http://dx.doi.org/10.1109/52.469757`.

[25] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.

[26] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, Apr. 2002. ISSN 1049-331X. . URL `http://doi.acm.org/10.1145/505145.505149`.

[27] G. E. Krasner and S. T. Pope. A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, Aug. 1988. ISSN 0896-8438. URL `http://dl.acm.org/citation.cfm?id=50757.50759`.

[28] J. Lemon. *kqueue (2)*. FreeBSD, 2008.

[29] T. M. H. Reenskaug. Thing-Model-View-Editor – an Example from a planningsystem. http://heim.ifi.uio.no/ trygver/1979/mvc-1/1979-05-MVC.pdf, May 1979. URL `http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf`.

[30] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 178–187, New York, NY, USA, 2000. ACM. ISBN 1-58113-206-9. . URL `http://doi.acm.org/10.1145/337180.337201`.

[31] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-specific Languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005. ISSN 0360-0300. . URL `http://doi.acm.org/10.1145/1118890.1118892`.

[32] Microsoft. Wunderlist. URL `https://www.wunderlist.com/home`.

[33] B. A. Myers. User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.*, 2(1):64–103, Mar. 1995. ISSN 1073-0516. . URL `http://doi.acm.org/10.1145/200968.200971`.

[34] B. A. Myers and M. B. Rosson. Survey on User Interface Programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 195–202, New York, NY, USA, 1992. ACM. ISBN 0-89791-513-5. . URL `http://doi.acm.org/10.1145/142750.142789`.

[35] B. A. Nardi and J. R. Miller. The Spreadsheet Interface: A Basis for End User Programming. In *Proceedings of the IFIP TC13 Third Interational Conference on Human-Computer Interaction*, INTERACT '90, pages 977–983, Amsterdam, The Netherlands, The Netherlands, 1990. North-Holland Publishing Co. ISBN 0-444-88817-9. URL `http://dl.acm.org/citation.cfm?id=647402.725609`.

[36] T. Neward. The Vietnam of Computer Science . URL `http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx`.

[37] D. Notkin, D. Garlan, W. G. Griswold, and K. J. Sullivan. Adding Implicit Invocation to Languages: Three Approaches. In *Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software*, pages 489–510, London, UK, UK, 1993. Springer-Verlag. ISBN 3-540-57342-9. URL `http://dl.acm.org/citation.cfm?id=646897.710010`.

[38] S. Oney, B. Myers, and J. Brandt. InterState: A Language and Environment for Expressing Interface Behavior. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 263–272, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3069-5. . URL `http://doi.acm.org/10.1145/2642918.2647358`.

[39] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT SOFTWARE ENGINEERING NOTES*, 17:40–52, 1992.

[40] K. W. Piersol. Object-oriented Spreadsheets: The Analytic Spreadsheet Package. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 385–390, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. . URL `http://doi.acm.org/10.1145/28697.28737`.

[41] H. Samimi, A. Warth, M. Eslamimehr, and A. Borning. Constraints As a Design Pattern. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 28–43, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3688-8. . URL `http://doi.acm.org/10.1145/2814228.2814244`.

[42] M. Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *Selected Papers from the Workshop on Studies of Software Design*, ICSE '93, pages 17–32, London, UK, UK, 1996. Springer-Verlag. ISBN 3-540-61285-8. URL `http://dl.acm.org/citation.cfm?id=645540.657852`.

[43] M. Shaw and P. C. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings of the 21st International Computer Software and Applications Conference*, COMPSAC '97, pages 6–13, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8105-5. URL `http://dl.acm.org/citation.cfm?id=645979.676005`.

[44] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[45] I. E. Sutherland. Sketchpad: A Man-machine Graphical Communication System. In *Proceedings of the May 21-23, 1963, Spring Joint Computer Conference*, AFIPS '63 (Spring), pages 329–346, New York, NY, USA, 1963. ACM. . URL `http://doi.acm.org/10.1145/1461551.1461591`.

[46] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009. ISBN 0470167742, 9780470167748.

[47] B. T. Vander Zanden, R. Halterman, B. A. Myers, R. McDaniel, R. Miller, P. Szekely, D. A. Giuse, and D. Kosbie. Lessons Learned About One-way, Dataflow Constraints in the Garnet and Amulet Graphical Toolkits. *ACM Trans. Program. Lang. Syst.*, 23(6):776–796, Nov. 2001. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/506315.506318`.

[48] B. T. Vander Zanden, R. Halterman, B. A. Myers, R. Miller, P. Szekely, D. A. Giuse, D. Kosbie, and R. McDaniel. Lessons Learned from Users' Experiences with Spreadsheet Constraints in the Garnet and Amulet Graphical Toolkits. 2002.

[49] B. T. Vander Zanden, D. Baker, and J. Jin. An Explanation-based, Visual Debugger for One-way Constraints. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, UIST '04, pages 207–216, New York, NY, USA, 2004. ACM. ISBN 1-58113-957-8. . URL `http://doi.acm.org/10.1145/1029632.1029670`.

[50] M. Weiher. Objective-Smalltalk: , 11 2010. URL `http://objective.st`.

[51] M. Weiher and C. Dowie. In-Process REST at the BBC. In C. Pautasso, E. Wilde, and R. Alarcon, editors, *REST: Advanced Research Topics and Practical Applications*, pages 193–209. Springer New York, 2014. ISBN 978-1-4614-9298-6. . URL `http://dx.doi.org/10.1007/978-1-4614-9299-3_11`.

[52] M. Weiher and R. Hirschfeld. Polymorphic Identifiers: Uniform Resource Access in Objective-Smalltalk. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 61–72, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2433-5. . URL `http://doi.acm.org/10.1145/2508168.2508169`.