Citrus: A Language and Toolkit for Simplifying the Creation of Structured Editors for Code and Data

Amy J. Ko and Brad A. Myers
Human Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213
ajko@cs.cmu.edu, bam+@cs.cmu.edu

ABSTRACT
Direct-manipulation editors for structured data are increasingly common. While such editors can greatly simplify the creation of structured data, there are few tools to simplify the creation of the editors themselves. This paper presents Citrus, a new programming language and user interface toolkit designed for this purpose. Citrus offers language-level support for constraints, restrictions and change notifications on primitive and aggregate data, mechanisms for automatically creating, removing, and reusing views as data changes, a library of widgets, layouts and behaviors for defining interactive views, and two comprehensive interactive editors as an interface to the language and toolkit itself. Together, these features support the creation of editors for a large class of data and code.

ACM Classification Keywords: H.5.2. [Information interfaces and presentation]: User Interfaces—toolkits.

General Terms: Human Factors, Languages

Keywords: Structured editing, interface builder, toolkit.

INTRODUCTION
With the advent of XML, developers are creating a growing number of graphical editors for structured data, including HTML editors, forms-based interfaces, visual programming environments, editors for source code, diagram editors, and even calendars, as well as editors for specialized data types designed for particular needs. While many toolkits [1, 10, 14, 18] exist that can be used to create these editors, several common implementation tasks involved in building these editors, such as expressing constraints, validating user input, responding to changes of the data, and synchronizing the models and views, are still cumbersome and error prone in many circumstances.

In this paper we describe Citrus (**C**reating **I**nteractive **T**ools for **R**eshaping and **U**tilizing **S**tructure), a new language and user interface toolkit designed to simplify the creation of editors for structured data and code. Citrus distills what have traditionally been toolkit-level features into first-class language features, and offers toolkit-level support for features that have previously had to be implemented from scratch. The result is a simple, expressive and powerful language and accompanying user interface toolkit that is specifically designed for creating model-view-controller-based direct-manipulation editors for highly structured data.

UIST'05, October 23–27, 2005, Seattle, Washington, USA.

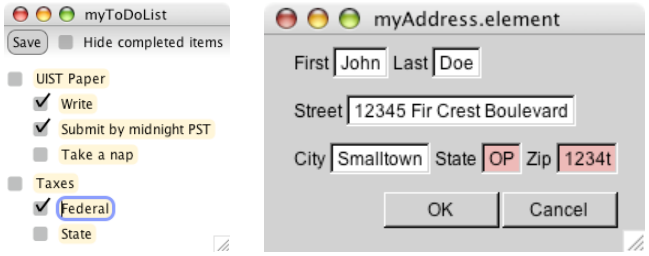

Figure 1. To-do list and address editors created with the Citrus language and toolkit.

In addition to including well-known features such as constraints and change notifications, Citrus includes language-level support for restrictions on primitive and aggregate data and for automatically synchronizing models and views as data changes. It offers toolkit-level support for drag and drop interactions and provides auto-completion to text fields with little to no additional code. Using these features and others, the Citrus language can describe a wide range of data types, from simple data such as addresses, to complex executable programs, as well as aid in the creation of interactive editors for these data types. Citrus can even be used as an alternative to parser-generators such as YACC by simplifying the creation of graphical structured editors for programming languages that will parse the resulting code as a programmer types.

In the next section we offer an example of building the to-do list editor seen in Figure 1. We then describe the features of the Citrus language and user interface toolkit in detail, followed by several examples of larger editors built with Citrus, including Citrus's own specification editor and interface builder, a flow chart editor, and a prototype of a Java programming environment. We end with a discussion of related work and some brief conclusions.

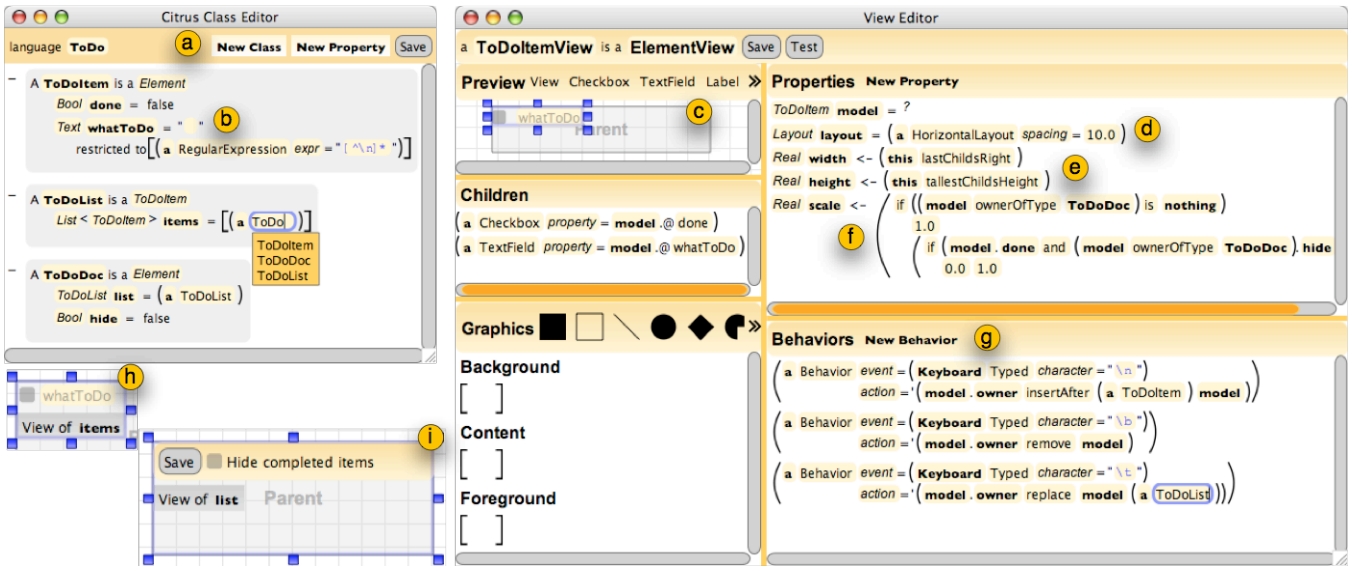

Figure 2. The Citrus specification editor (left) and interface builder (right), used to create Citrus editors.

CREATING A TO-DO LIST

Creating a Citrus editor involves writing a specification of the classes of data to be edited and then designing views for each class. As an alternative to using Citrus's textual syntax and parser, Citrus also provides a graphical editor for each step, as seen in Figure 2. All of the code for creating an editor can be specified interactively using these tools.

As our example, we will create the to-do list editor seen in Figure 1. We begin by defining the *model,* creating three classes by dragging `New Class` from the toolbar at (a) in Figure 2. We name them `ToDoItem`, `ToDoList`, and `ToDoDoc`. Each is declared with the syntax "`a Name is a Superclass`," followed by a list of declarations of the instance variables, which Citrus calls *properties*, of the form "`Type name=expression restricted to [...]`." For example, `ToDoItems` have two properties, which we dragged from the toolbar: `done`, a Boolean, and `whatToDo`, a text string that has the regular expression *value restriction* at (b), disallowing carriage returns. In Citrus syntax, `[]`'s denote list creation, `()`'s denote method calls and instantiations, and `<>`'s denote type arguments for generic classes (where `List<ToDoItem>` is a list of `ToDoItems`). A '.' gets a property's value and @ gets a property itself.

Next, we design *views* for the three classes, using the Citrus interface builder on the right in Figure 2. Rather than creating static displays like conventional builders, Citrus defines a view *class* that is instantiated for each instance of data of a particular type. We define a view by choosing views for each of a class's properties. For example, for `ToDoItemView`, shown in Figure 2, we drag a checkbox and text field from the toolbar at (c). As we do so, the interface builder chooses an unused property in `ToDoItem` of the appropriate type. These can be changed, if necessary, in the "Children" panel, which shows the children that will be instantiated for each instance of `ToDoItemView`.

To arrange the checkbox and the text field, we type the expression `(a HorizontalLayout spacing=10.0)` for the `layout` property at (d). By default, the top and left properties of all views are constrained to the result of the `getLeft` and `getTop` methods of their parent's `layout`. We then type expressions for `width` and `height` at (e), fitting the view to its layout. The `<-` indicates that these properties are *constrained*, rather than initialized to these expressions.

To hide completed items in the editor, we write a constraint on the `scale` property at (f) that uses the `done` property defined in `ToDoDoc`. It returns `0.0` if the `done` property of the `ToDoItem` and `hide` property of the `ToDoDoc` that "owns" the `ToDoItem` are both true, and `1.0` otherwise.

To enable the insertion and removal of `ToDoItems`, we add *behaviors* to the view class by dragging them into the behavior panel at (g). When a carriage return is typed, a new `ToDoItem` will be inserted after the `ToDoItem` that the view represents; when a backspace is typed, the `ToDoItem` that the view represents will be removed; and when a tab is typed, the `ToDoItem` will be replaced with a `ToDoList`.

The views for `ToDoList` and `ToDoDoc`, shown at (h) and (i), each have a placeholder that translates a property into an appropriate view for the property's type. In this case, `items`, a `List`, is translated into a `ListView`, which updates itself as its list changes; `list`, of type `ToDoList`, is translated into an instance of the `ToDoListView` class that we defined. The "Save" button calls `write` on the `ToDoDoc`, which uses a built-in routine to serialize the data in a straightforward XML format.

Building this editor with a more traditional toolkit would have required several lines of code to synchronize models and views, to generate new views as items are inserted, to validate `whatToDo`, to save the data, and to implement constraints. Citrus manages all of these automatically.

OVERVIEW

We have built several powerful applications using the Citrus language and user interface toolkit, including Citrus's own specification editor and interface. The key to Citrus's power and simplicity lies in its language and toolkit-level support for implementation tasks that typically must be done manually in other toolkits. In the following sections we describe this support in detail.

THE CITRUS SPECIFICATION LANGUAGE

Citrus is an object-oriented, statically-typed and interpreted language, with support for generic classes and reflection. Like other object-oriented languages, Citrus programs consist of classes that declare a list of named instance variables and methods. We refer to an instance of a Citrus class as an *element* to reinforce the notion that they generally represent data (just as XML elements represent data); for the same reason, we call the instance variables *properties*. In addition to basic features such as inheritance and method overriding, Citrus also allows property overriding, which lets programmers override property's default values, constraints and types, either by subclassing or when instantiating an element. Everything in Citrus is an element, including the primitives (Booleans, integers, floating-point values, text strings, lists, and sets), the implementation of the properties themselves, as well as every other data structure described throughout this paper. Citrus classes can be defined using the Citrus language, using the interactive editor (Figure 2 left), its textual syntax, or by writing Java code (this last technique has only been necessary for bootstrapping the system).

Expressions

Citrus expressions are of the form `(context method arguments)`, except for the syntax for list creation and conditionals. Methods are typically defined in a Citrus class, but Citrus also supports higher-order anonymous methods and quoted expressions, allowing code to be used as data (for examples, see the quoted expressions in the behaviors shown in Figure 2). Although this makes static type checking impossible in many cases, it adds a level of flexibility found in dynamically typed languages such as Lisp. Citrus also supports special methods for getting and setting a property's value and getting a handle to the property itself (e.g., `whatToDo` refers to the property's value and `@whatToDo` refers to the property itself).

Properties

The value stored in each Citrus property is a pointer to a single Citrus element. Unlike pointers in other languages, Citrus pointers can either *own* or *refer to* the element that they point to. This allows Citrus elements to represent both cyclic and acyclic graphs of elements. Each element also has a reference to the property that owns it and to the set of properties that refer to it. This is used when an element needs to know its context (for example, (f) in Figure 2 shows how to determine the `ToDoDoc` that owns the `ToDoItem` using the `ownerOfType` method).

Each property declaration has a *value expression*, which is must be supplied as either an initial value (using the symbol `=`), a constraint (using the symbol `<-`), or as a parameterized value that must be passed during instantiation (using the symbol `?`). Value expression can be overridden in instantiation expressions using the property's name. For example, the expression `(a ToDoItem done=true whatToDo<-"work")` creates a new `ToDoItem`, overriding the `done` property's initial value with true, and constraining `whatToDo` to the text string "work." This avoids the need to subclass when the only changes to an instance are to initial values and constraints. Constraints on properties can also be set at runtime.

Because a property's value expression may depend on other as yet uninitialized properties, properties are initialized lazily (unless they are parameterized, in which case they are initialized immediately). For properties that are constrained, Citrus lazily evaluates the constraint as dependencies change at runtime. Constraints are based on Amulet's dynamic one-way constraint algorithm [20], and allow arbitrary expressions, including pointer variables (for example, a reference to a view's parent's width). Each property maintains bookkeeping about the incoming and outgoing edges in its constraint's dependency graph. Cycles are detected at runtime, as in Amulet's constraint system, and use the "once around" method for breaking a cycle.

Property and Element Restrictions

Property restrictions specify a property's legal values. Unlike other toolkits with support for validation, such as XML Schema data types, Citrus's property restrictions can be expressed using *any* Boolean-valued single-argument method, including those that refer to other properties. Citrus automatically records any dependencies in the method at runtime and revalidates the property as they change. For example, Citrus' scroll bars use property restrictions to require that the bar remains within the bounds of the scroll bar track, and that the bar stay sufficiently tall and wide for interaction. Citrus also provides several common restrictions, such as ranges for continuous values and regular expressions for text strings (as seen in our `ToDoItem`'s `whatToDo`). A property's validity can be requested, and elements can also be notified when a property's validity changes (this is discussed shortly).

To allow for flexibility in the enforcement of property restrictions, each restriction has an `allowInvalid` flag; if set, invalid values are assigned, but if not set, trying to set an invalid value causes the property to retain its old value. For example, in our to-do list editor, the regular expression's `allowInvalid` is false by default, disallowing carriage-returns. In our Java editor, we use a regular expression to define a valid Java identifier, but we set the flag to true to allow temporarily invalid identifiers as programmers are typing.

Many property restrictions can be represented as constraints (for example, a constraint containing min and max could be used in place of boundary restrictions for Citrus's scroll bars), but there are several advantages to their separation. For example, a zip code property might be restricted to valid zip codes (as in the address dialog in Figure 1), but would not be constrained to a *particular* zip code. This separation also allows property restrictions to define a `validValues` method that computes a finite set of legal values for a property. Our Java editor defines this method to compute the set of valid names for variable references based on Java's scoping rules. Citrus text fields take full advantage of this feature to support auto-completion.

Just as properties can declare property restrictions, a class can declare any number of *element restrictions*, which are conditions that determine an element's validity. For example, an element that represents a form might require that a specific set of fields be completed and that if a particular field is filled another field must be as well.

Listeners and Notification

It is often necessary to take some action when a property or element changes. This is typically achieved by adding and removing *listeners* to data. Citrus provides language-level support for such notification mechanisms, allowing notification about events on a single property's value or of changes to any element it refers to indirectly (called "deep monitoring" by Rodham and Olsen [17]). Programmers simply write a declaration of the form `when` *`expression event action`*. At runtime, Citrus automatically adds and removes a listener to the property or element that *`expression`* evaluates to as dependencies in the expression change. When the event of interest occurs on the current property or element of interest, the *`action`* is executed. The `willChange` and `changed` events are sent before and after a property or element changes. The `outOfDate` event is sent when dependencies in property's constraint or an element's restriction change. The `validityWillChange` and `validityChanged` events are sent before and after a property is set to a value that changes its validity one when an element's restriction is or is no longer violated. This is used to make the state and zip code fields in Figure 1 change to red when invalid. The `cycle` event is sent when a cycle is detected in the dependency graph of the property's constraint. This allows programmers to respond in an appropriate way to the cycle, possibly halting the editor, relaxing the constraint, or notifying the user.

Serialization

We take advantage of Citrus's common data format to provide general support for serializing any Citrus element into XML. If a property is constrained or parameterized, it is not serialized under the assumption that it can be recovered later. Views of each element are serialized with their model in order to save any unrecoverable view state, such as absolute positions or collapsed/expanded state.

Specifying Languages

Many of the features of the Citrus language are suitable for creating graphical editors and interpreters for programming languages. For example, each Citrus element is roughly equivalent to a node in an abstract syntax tree, where the properties of the element correspond to the attributes on the node. Property restrictions can be used to express static scoping rules for references in a language; we have used these to specify some of the static scoping rules for Java. A collection of Citrus classes can be used to define the equivalent of a context-free grammar. Classes can also define an `evaluate` method, which "executes" an element. This is used, for example, to execute a form when its okay button is pressed, possibly by sending a database query consisting of the values of each property stored in the element. We have even used `evaluate` methods to implement the Citrus interpreter.

THE CITRUS USER INTERFACE TOOLKIT

As with the Citrus language, the central design goals for the Citrus user interface toolkit were flexibility and expressiveness. Much of this was achieved by simply implementing the toolkit using the Citrus language. For example, views automatically repaint themselves when any of their properties change, simply by listening to `changed` and `outOfDate` events on themselves. In addition, flexibility and expressiveness were achieved by making all toolkit primitives dynamic and by providing a rich set of widgets that interact closely with Citrus language features.

Views and Element Views

`Views` are the basis of the Citrus user interface toolkit, acting as containers for other views and defining a local coordinate system. The majority of views in an editor, however, are `ElementViews`, which represent the current value of a particular property. The rationale behind these views was that when writing code to manipulate a model, programmers should not have to be concerned with managing and updating the model's corresponding views as the model changes.

To accomplish this, `ElementViews` listen to changes in the property that they represent, and on a change, either update themselves or replaces themselves with a view of their property's new element. To illustrate, consider Figure 3, which portrays the elements and views involved in representing the `ToDoItemView` from our earlier example. The `ToDoItemView` listens to the property that currently points to the `ToDoItem` that the view represents. When this property changes, the `ToDoItemView` replaces itself with a view of the new element pointed to. For example, the `replace` statement in the behavior at (g) in Figure 2 sets the property that currently owns the `ToDoItem` to a new `ToDoList`, causing the `ToDoItemView` to replace itself with a new `ToDoListView`. The checkbox and text field listen to changes in the `ToDoItem`'s `done` and `whatToDo` properties and modify themselves on each change.

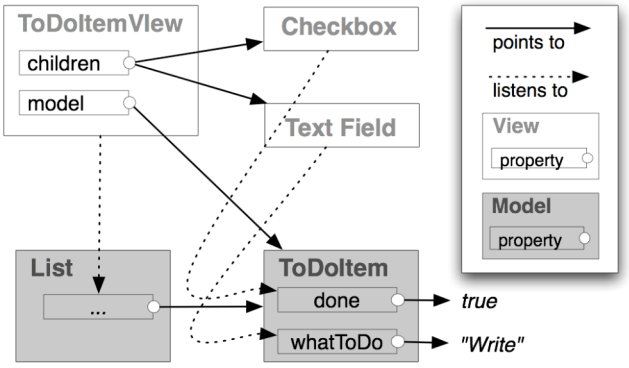

Figure 3. Entities involved in representing a `ToDoItemView`. Each view listens to changes the in a particular property's value. When their properties change, the checkbox and text field update themselves, whereas the `ToDoItemView` replaces itself with a view the property's new value.

To use this translation mechanism, the programmer simply calls the `toView` method on a property. For example, the `ToDoDocView` in Figure 2 uses the expression `(model.@list toView)` to create an appropriate view for it's `ToDoDoc`'s `list` property. Citrus uses a global repository that maps Citrus classes to a corresponding view class, which is then instantiated. The view manages every change from then on, so that the programmer can focus on modifying the model. Because Citrus classes can be specialized (for example, lists can be declared to contain a particular class of element), this repository distinguishes between generic views of Citrus classes, and more specific views for more specific types. For example, as implemented, our to-do list editor implicitly used generic `ListViews` since a more specific type of `ListView` was not defined. To do so, we would define a subclass of `ListView` that specifies its `model` property's type as `List<ToDoItem>`, and then proceed to add custom interactions, appearance, and layout specifically designed for lists of `ToDoItems`.

In addition to each view class representing a particular class of element, it is also specified as one of three varieties. *Ownership* views (the default) represent a property that *owns* the element it points to (as defined in the Citrus language section). All of the views created for the to-do list editor were ownership views. *Reference* views represent properties that *reference*, rather than *own* the element they point to. This is used, for example, when a property refers to an element shown somewhere else onscreen, where it makes more sense to use the element's name or some other summary, rather than its ownership view. *Nothing* views are used to represent properties that point to nothing. Unlike most languages, where a "null" pointer is always represented in the same way, Citrus allows a different view of nothing for each Citrus class. Citrus supplies generic ownership, reference, nothing views when none are specified for given class.

Layout

Layout tends to be a point of contention in many toolkits, and thus, in keeping with our goal of flexibility, we tried to offer several ways of defining view layouts. For full flexibility, the programmer can write custom constraints for each view's top–left position. In many cases, however, the same constraint is applied to all of the children of a view. For example, in a vertical left-justified list layout, each view's left is always zero and its top is always its previous sibling's bottom (except for the first). Rather than requiring programmers to repeat these constraints, Citrus provides `Layout` elements, which contain two methods to define the top and left properties of each child of the associated view. By default, a view's top-left position is constrained to the values returned by its parent's layout element. Citrus offers several pre-defined and highly parameterized layouts, including vertical, horizontal, centered, and horizontal flow layouts. Custom layouts are made by simply subclassing `Layout` and defining its two methods.

It is also straightforward to write layout algorithms, rather than use constraints or the layout elements described above. These can be invoked in a number of ways; for example, a listener could be added to a view's children property so that any time a child is added or removed, the algorithm could be invoked. Or, a widget could be provided that invokes the algorithm. We later give an example of this when describing the connected graph editor.

Graphic Objects

A view's appearance is determined by background, content, and foreground lists of retained `GraphicObjects`. Citrus provides primitives including rectangles, lines, polygons, circles, ellipses, and text, and an aggregate graphic object, which allows custom shapes to be defined from the addition or subtraction of primitive shapes and other aggregates. Each `GraphicObject` has several properties; for example, rectangles have a `roundedness` and `filled` property, lines have a `thickness` property, and all have primary and secondary colors and basic support for gradients. Each property can be set and constrained when a `GraphicObject` is instantiated, avoiding the need to subclass them.

To reduce the need for constraints, each graphic object is rendered relative to a view's top, left, right, and bottom, and includes both pixel and proportional offset properties. For example, to paint a 1-pixel border around a view, a graphic object's offsets would be (–1, –1) for the top-left and (1, 1) for the bottom-right; a graphic object could be centered and half the size of a view by specifying its proportions to be 25% for the top, left, bottom, and right.

Programmers can specify a view's default graphic objects by dragging graphic objects from the toolbar in the interface builder to one of the three lists of graphic objects (seen in the bottom left of the interface builder in Figure 2). Graphic objects can also be added and removed at runtime using the standard list insert and remove methods.

Behaviors and Devices

Citrus uses an event-based input model that was designed to be flexible and dynamic. Events are generated by `Device` elements, such as a mouse and keyboard, which each declare a list of sub-devices and a list of events. For example, the `Mouse` device has several button devices, a scroll-wheel device, and a pointer device; each mouse button specifies `pressed`, `released`, `clicked`, and `double-clicked` events. Each device has several additional properties: for example, the mouse pointer device has a position and a reference to the view currently picked by the pointer; keyboard keys have a `down` property. Because all devices and their properties are elements, they can be listened to and referenced in constraints. New devices can be defined with Java to support alternative input techniques, such as gesture or pen-based interactions.

To respond to events, views have a list of `Behavior` elements, which react to a particular type of device event with particular parameters by executing an arbitrary Boolean-valued expression. Behaviors can consume events in two ways. By default, if an action returns true, the event is consumed only after all behaviors in a view that react to a particular event are executed. This allows for extensibility; for example, even though Citrus text fields consume the keyboard's typed events, subclasses can still respond to the event in additional ways. Behaviors can also be set to immediately consume an event, overriding behaviors in a view that would have otherwise reacted to the event. Behaviors can be added and removed at runtime as part of other behaviors. For example, the toolbar widgets used in the specification editor and interface builder add `Draggable` behaviors (described shortly) to views that are not draggable by default, removing them afterwards.

Pointer and Keyboard Focus

The mouse pointer maintains a pointer focus, allowing views to temporarily redirect all mouse events to a particular view. For example, Citrus's sliders obtain the pointer focus while being dragged, and release focus on the mouse button released event. Views specify a `Shape` element for mouse pointer picking purposes.

Keyboard focus in Citrus works in the same way that it does in most user interface toolkits; each window has a keyboard focus, and all keyboard input goes to the current view in focus in the active window, or one of its parents. However, the *movement* of the keyboard focus is different than that in most toolkits, which typically specify a focus "ring", relatively independent of the layout of the interface. Instead, Citrus views define several methods that determine the spatially nearest focusable view left, right, above, and below itself. These methods are used to map the keyboard's arrow keys to movements of the keyboard focus to adjacent focusable views. Citrus also offers methods to mimic text-editors. For example, pressing the right arrow key at the end of a line wraps to the beginning of the next line. This corresponds to a depth-first search through the view hierarchy for focusable views that are laid out top-to-bottom, left-to-right. By default, all of these methods search for the deepest focusable view in a hierarchy, because most keyboard input goes to text widgets. This also avoids ambiguity: if a focusable view that contained other focusable children obtained focus, it would be unclear whether pressing the down arrow would give focus to the next focusable sibling or child.

Drag and Drop Behaviors

Because all Citrus views have the same programming interface, we were able to design general drag and drop behaviors for use in any Citrus view. They are implemented as aggregate behaviors that respond to the various mouse pointer events associated with dragging. A view that includes a `Draggable` behavior, for example, is hoisted when clicked, moved as the mouse pointer moves, and returned to its original location when released over an invalid location. When released over a view of a property with a compatible type as the model of the view being dragged, the behavior either sets, references, or duplicates the model being dragged, depending on its `action` property. A view that includes a `Droppable` behavior reacts to the `draggedOver` and `droppedOver` events by checking if the view that is being dragged represents an element of the expected class by the droppable view's property, and if so, setting the property to the element being dragged. Citrus also provides an `Insertable` behavior for list views. These have been sufficient for all the editors we have developed.

Animation

When setting a property, a `Transition` element can be supplied to animate the change. Transitions, as in prior work [6, 11], consist of a method to map the property's value based on elapsed time and a pacing function. In addition to single value transitions, Citrus also supports sequenced animations via two animation constructs called `DoInOrder` and `DoTogether` (borrowed from the Alice programming system [3]). By default, behaviors finish executing before returning control to the event loop; animations allow for behaviors to execute over time. Citrus uses animations in many of its pre-defined behaviors. For example, when a view is dropped over an invalid location, it is animated to its original position.

Styles

Each editor has a `Style` element, which is a repository for fonts, colors, graphics, line spacing, behaviors and any other shared elements. The default style provided by Citrus includes defaults for all of the pre-defined widgets in the toolkit. Each widget parameterizes its colors, behaviors, and graphics objects using the style, allowing every aspect of the widget to be modified, except for the widget's view hierarchy. Programmers can subclass styles to include additional properties. For example, we created a custom style for the address editor seen in Figure 1, requiring the several new colors and graphics for text fields and buttons.

EXAMPLES

In the following sections we illustrate the range and expressiveness of Citrus by describing the implementation of four diverse editors built with its language and toolkit.

THE CITRUS SPECIFICATION EDITOR

One approach to building an editor for the Citrus language would have been to create a parser and use a regular text editor to edit specifications. However, adding programming support to the editor, such as auto-completion and immediate feedback about errors, would have required significant effort and incremental parsing algorithms.

Creating a code editor with these features using Citrus was easy. We simply defined several Citrus classes to implement the specification language itself, including classes for classes, property declarations, property restrictions, primitives, and so on. Since we had no editor to begin with, these were bootstrapped using our Java API for defining Citrus classes. We then defined views for each Citrus class. The only views we specified beyond the *ownership* view of each class were several types of views of lists containing particular types.

Citrus's drag and drop behaviors greatly simplified the creation of the editor. By simply adding `Draggable`, `Droppable`, and `Insertable` behaviors to the appropriate views, no additional code was necessary to implement the interactions in the editor. `Duplicator` widgets were used to create new code; when clicked, they duplicate a specified element and create a corresponding view. To support deletion, views responds to the backspace key by replacing themselves with nothing, or in the case of elements in lists, removing themselves from the list that owns them.

EXAMPLE: THE CITRUS INTERFACE BUILDER

The Citrus interface builder, seen in Figure 2, edits any subclass of `View`. Designing a view involves defining value expressions and restrictions for each property. For example, modifications to the lists of children, graphics, and behaviors are simply modifications to the list creation expressions assigned as each property's value expression. Nearly everything can be defined interactively, since interactive views were defined for each part of the class definition, including layouts, graphics objects, children, and so on. The only details that must be specified with code are value expressions and behaviors, and even then, they can be created using the same drag and drop behaviors used in the specification editor. The interface builder also uses Citrus text fields, which have general support for auto-completion by asking their properties for their set of valid values.

A key feature of the builder is its ability to provide immediate feedback by showing an instance of the class being defined. The editor uses a listener to respond to changes in each declaration's value expression by updating the instance with the new expression. Because Citrus incrementally updates the expression's validity, the editor is notified when it is unsafe to execute the expression.

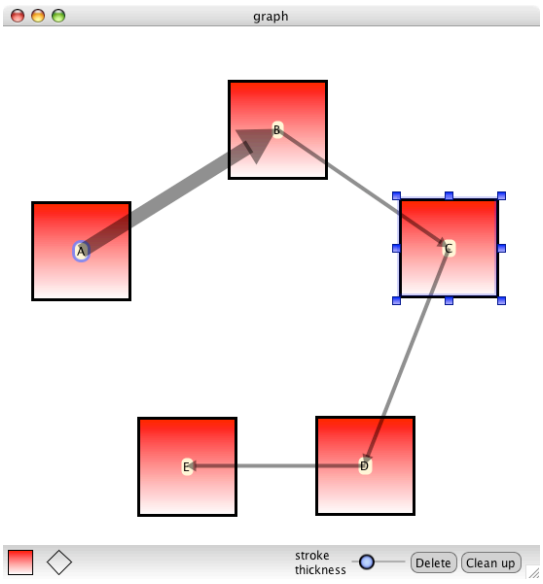

Figure 4. A simple connected graph editor, supporting multiple interaction techniques for creating nodes and arrows, control over the line stroke thickness of any object, and automatic "neatening" of the layout.

A BENCHMARK GRAPH EDITOR

The second author has proposed several benchmarks for comparing the expressiveness and usability of user interface toolkits, and many people have implemented these benchmarks with dozens of toolkits [12]. The *graph editor* benchmark, which allows the creation of box and arrow diagrams, is particularly suitable for evaluating Citrus. The benchmark has six requirements: (1) the boxes are labeled with text; (2) each box is attached by a line with an arrow to the box that was previously drawn; (3) the layout is under the user's control; (4) the boxes can be resized and moved; (5) the stroke thickness of the boxes and lines can be changed; and (6) the boxes and arrows can be deleted. The second author was able to implement this benchmark in Amulet [13] in about 90 minutes using 212 lines of code.

Our Citrus implementation of the benchmark is seen in Figure 4. Every view in the editor is custom made, except for the selection handles, slider, and buttons. Nodes are created by clicking and dragging on the canvas. The stroke thickness of nodes and arrows can be modified with the slider at the bottom right, whose property is constrained to the stroke property of the graphic object of the view currently selected. All of these behaviors were implemented with the Citrus interface builder.

Our implementation adds several other features beyond those required by the benchmark. For example, nodes can also be created by dragging them from the toolbar. Each arrow has a clickable region that is a line shape with a thickness twice that of the line, allowing for easier clicking. The flow chart editor has a "Clean up" button that invokes a method that arranges the nodes in the editor in a grid.

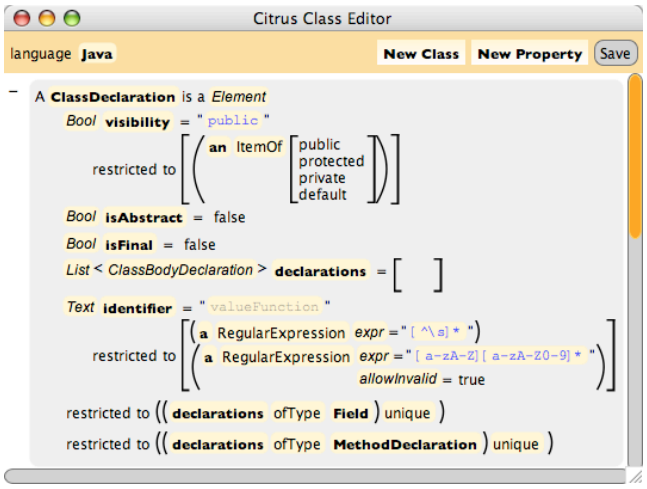

Figure 5. A partial specification of the Java programming language's class declaration construct.

Even with the additional features, the first author was able implement the benchmark in just 30 minutes and 63 lines of code. This included 11 lines for the specification, and 25 lines of custom behaviors, 12 lines of pre-defined layouts and constraints, 4 lines of custom constraints for the arrows, and 11 for the "clean up" layout method. This is despite the fact that Citrus is not even focused on creating applications of this type.

A JAVA EDITOR

Because Citrus classes describe a structure similar to an abstract syntax tree, it is easy to map a programming language's syntax to Citrus classes. For example, the formal syntax for a Java class declaration is:

```
ClassDeclaration::
        ClassModifiers class Identifier
        Super Interfaces ClassBody
```

This syntax is represented as the Citrus class seen in Figure 5. The `ClassModifiers` non-terminal is represented as three properties: a text string, restricted to "public," "protected," "private," and "default," and two Booleans representing the abstract and final keywords. A text string represents the class declaration's identifier, which may not contain whitespace, but may temporarily contain non-alphanumeric characters. A list of `ClassBodyDeclarations` represents the `ClassBody` non-terminal.

Because this Citrus class separates the specification of a Java class declaration from its visual representation, we can represent a class declaration in the most appropriate way for the data. For example, Figure 6 shows our prototype Java editor's view of a class declaration, laid out from left to right and top to bottom. It places a scroll view around the class body part, allowing the class header to always remain at the top of the screen as context. Alternative views could easily be created; for example, a view with an absolute positioning layout would allow programmers to visually arrange the class body declarations in whatever way was

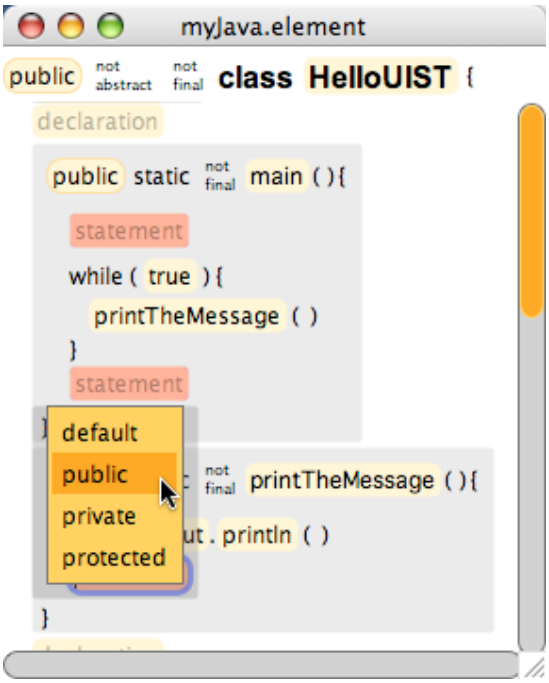

Figure 6. A prototype of a Java editor, created using Citrus. The editor supports incremental error checking and code completion.

most appropriate for their task, grouping declarations together, or placing related declarations side by side for comparison. Such a view would be difficult to implement in a text editor, but here would only require a single change to the class declaration view's layout property.

Because property restrictions were used throughout the specification to define valid identifiers for classes, methods, and variables, and to define Java's static scoping rules, every text field and pop-up menu in the prototype automatically supports auto-completion and immediate feedback about errors (as seen in Figure 6). The element restrictions at the bottom of Figure 5 use the `List` class's `ofType` and `unique` methods to require that the `Fields` and `MethodDeclarations` in the `declarations` list are not equivalent to any other declaration in the list, based on each class's definition of equivalence. `Fields` define equivalence by name and `MethodDeclarations` define it by their method signature.

In addition to drag and drop interactions, which are supported throughout the Java editor, we are currently focusing on improving the keyboard-based interactions. We have implemented several novel types of edits that have the potential to improve the flexibility of structured editors. For example, typing "if" and a space inside a statement replaces the empty statement with an `if` statement, just like typing tab replaced a `ToDoItem` with a `ToDoList` in our to-do list editor. Though the keyboard-based interaction techniques have potential, we feel they still need considerable work before they will be smooth enough for regular use.

DISCUSSION AND RELATED WORK

Each design choice we made for Citrus focused on providing a generic, semi-automated solution for specifying model-view-controller based graphical editors. Because of this focused intent, one way to think of Citrus is as a "scripting" language, like Mathematica, Visual Basic, Perl, PHP, Tcl and others. This perspective offers many insights into Citrus's scope and limitations. For example, because it focuses on optimizing flexibility and expressiveness for a particular class of software architectures, Citrus is less suitable for applications that are not easily architected with models and views; Citrus may therefore not be the right language for exploring AI or mathematical computations. Furthermore, just as with other scripting languages, the applications that Citrus helps develop *can* be developed with more general languages such as Java, but with more code and hassle. Citrus does, however, share the same advantages as other scripting languages including being platform independent and supporting more rapid prototyping than compiled languages.

There are several alternative design choices that we could have made throughout the language and toolkit. These are best described by comparing Citrus to related tools, toolkits and languages that have similar and overlapping goals.

Citrus is closely related to previous work in syntax-directed and structured editors [10, 15, 16, 19], which derive views of data from the data itself. Many of these systems, most notably the GNOME and MacGnome environments [10] have the explicit goal of supporting partial automatic generation of structured editors from syntax grammars. However, the resulting editors typically have generic views and fixed interaction techniques. Citrus takes the approach of instead simplifying the development of custom, hand-designed structured editors. Many of the toolkits designed for building structured editors also support the specification of static semantics for code, but using a separate language. Citrus allows programmers to use the same language, and specify the semantics in the same place as the syntax.

Also related is work on "syntax-recognizing" editors [1, 2, 4, 18], in which a document is derived from the view, instead of the view from the document, typically via parsing. Although these toolkits often result in editors with more flexible keyboard-based interaction techniques, they do so by limiting the presentation of data largely to text. For example, the Proxima toolkit [18] restricts the layout of views to rows and columns. Citrus focuses on the alternative approach of allowing whatever presentation arrangement is most appropriate for the data being presented, even absolute positioning. This is potentially at the expense of the usability of its keyboard-based interaction techniques. To investigate this, we are currently designing new interaction techniques based on studies of Java programmers' use of unstructured text [7] that we hope will increase the flexibility of editors for highly-structured data such as code. We have begun to prototype the Java environment seen in Figure 6, in order to explore a class of programming tools that are difficult or impossible to implement with unstructured text, due to parsing ambiguities and limitations of the visual representation [8].

In recent years that has been a renewed interest in developing structured editors for end-user and novice programmers, including the Alice 3D programming system [3] and Scratch [9], which both support drag and drop interactions to construct code. Citrus is not a direct competitor to these systems, but rather, a language and toolkit that could simplify the development of such programming systems through its support for specifying languages and its generalized drag and drop behaviors.

The Citrus user interface toolkit shares many features with user interface toolkits such as Amulet [13] and SubArctic [5], including its support for constraints, graphics objects, and animation. Where Citrus differs is in its dynamic treatment of what are typically statically defined code in these toolkits, such as drawing, constraints, and layout, as dynamic, reusable, and sharable data. For example, Amulet's "graphical objects" combine positioning and appearance, where as Citrus separates the two, allowing layout and appearances to be more easily shared and modified at runtime.

Modern web tools, toolkits, and languages are also relevant. For example, XForms is an XML standard that helps separate the presentation from content for form-based web applications. The key difference from Citrus is that XForms focuses specifically on forms and transaction-based user interfaces. Citrus's property restrictions are similar to those in XML Schema datatypes, the difference being that Citrus's property restrictions can be defined with arbitrary expressions that can refer to any properties in the local context of their declaration. XML Schema types are restricted to a set of stereotypical restrictions and do not support dynamic references to other data. One central difference between Citrus and many of these web technologies is that it integrates several useful features into a single language; many XML-based languages and toolkits require programmers to learn several new languages.

Haystack [14], a platform for authoring semantic web applications, is similar to Citrus in many respects, offering customizable views for browsing, searching, and associating semi-structured data, a mechanism for choosing "view prescriptions" based on a data element's type, and a generalized data representation. Citrus and Haystack differ in the type of user activities they support: Haystack focuses on information management and searching applications, whereas Citrus focuses on information authoring and presentation (although both can support all of these activities to differing degrees). These different focuses result in different interaction techniques, data representations, and language support.

CONCLUSIONS

In addition to designing more pre-defined behaviors and widgets, we are currently focusing on improving the text-based interaction techniques of Citrus editors in the context of our Java editor. We hope to have the toolkit and editors ready for user tests and deployment in the near future. We have found the Citrus language and toolkit to be a flexible, expressive and powerful way to create graphical editors for structured data and code. We expect it to be increasingly useful as such data becomes more ubiquitous, and especially as end users become involved in creating and modifying structured data as part of their everyday work.

ACKNOWLEDGMENTS

We thank David Weitzman, James Fogarty, Jeff Nichols, Andrew Faulring, and Jeffrey Stylos for their input. This research was funded by the National Science Foundation under NSF grant IIS-0329090, and as part of the EUSES consortium under NSF grant ITR CCR-0324770. The first author is supported by an NDSEG fellowship.

REFERENCES

1. Ballance, R. A., Graham, S. L., and Vanter, M. L. V. d., The Pan Language-Based Editing System, *ACM Transactions on Software Engineering and Methodology*, 1, 1, 95-127, 1992.
2. Boshernitsan, M., Harmonia: A Flexible Framework for Constructing Interactive Language-Based Programming Tools., University of California, Berkeley Technical Report CSD-01-1149, 2001.
3. Dann, W., Cooper, S., and Pausch, R., Learning to Program with Alice: Prentice-Hall, 2003.
4. Horgan, J. R. and Moore, D. J., Techniques for Improving Language-Based Editors, *ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, 7-14, 1984.
5. Hudson, S. and Smith, I., Supporting Dynamic Downloadable Appearances in an Extensible User Interface Toolkit, *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Banff, Alberta, Canada, 159-168, 1997.
6. Hudson, S. E. and Stasko, J. T., Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions, *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Atlanta, GA, 57-67, 1993.
7. Ko, A. J., Aung, H., and Myers, B. A., Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing, *CHI '05: Human Factors in Computing*, Portland, OR, USA, (to appear), 2005.
8. Ko, A. J., Aung, H., and Myers, B. A., Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks, *International Conference on Software Engineering*, St. Louis, MI, to appear, 2005.
9. Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., and Resnick, M., Scratch: A Sneak Preview, *2nd International Conference on Creating, Connecting, and Collaborating through Computing*, Kyoto, Japan, 104-109, 2004.
10. Miller, P., Pane, J., Meter, G., and Vorthmann, S., Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University, *Interactive Learning Environments*, 4, 2, 140-158, 1994.
11. Myers, B. A., Miller, R. C., McDaniel, R., and Ferrency, A., Easily Adding Animations to Interfaces Using Constraints, *ACM Symposium on User Interface Software and Technology*, Seattle, WA, 119-128, 1996.
12. Myers, B. A., Altman, N., Amiri, K., Centurion, M., Chang, F., Chen, C., Derby, H., Huebner, J., Kaylor, R., Melton, R., O'Callahan, R., Tarpy, M., Unyelioglu, K., Wang, Z., and Warner, R., Using Benchmarks to Teach and Evaluate User Interface Tools, 1997. http://cs.cmu.edu/~amulet/papers/benchmarks.pdf
13. Myers, B. A., McDaniel, R. G., Miller, R. C., Ferrency, A., Faulring, A., Kyle, B. D., Mickish, A., Klimovitski, A., and Doane, P., The Amulet Environment: New Models for Effective User Interface Software Development, *IEEE Transactions on Software Engineering*, 23, 6, 347-365, 1997.
14. Quan, D., Huynh, D., and Karger, D. R., Haystack: A Platform for Authoring End User Semantic Web Applications, *2nd International Semantic Web Conference*, 2003.
15. Read, M. and Marlin, C., Generating Direct Manipulation Program Editors within the Multiview Programming Environment, *SIGSOFT Worshop*, San Francisco, CA, 232-236, 1996.
16. Reiss, S. P., Graphical Program Development with Pecan Program Development Systems, *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, 30-41, 1984.
17. Rodham, K. J. and Olsen Jr., D. R., Nanites: An Approach to Structure-Based Monitoring, *ACM Transactions on Computer Human Interaction*, To appear.
18. Schrage, M. M., "Proxima - a Presentation-Oriented Editor for Structured Documents," Utrecht University, 2004.
19. Teitelbaum, T. and Reps, T., The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, *Communications of the ACM*, 24, 9, 563-573, 1981.
20. Vander Zanden, B., Myers, B. A., Giuse, D., and Szekely, P., Integrating Pointer Variables into One-Way Constraint Models, *ACM Transactions on Computer Human Interaction*, 1, 2, 161-213, 1994.