CHAPTER 5 CREATING ALTERNATIVE DESIGN SOLUTIONS

5.1 ALTERNATIVES IN JUXTAPOSE

Design frequently alternates between divergent stages, where multiple different options are explored, and convergent stages, where ideas are selected and refined [55,66,135] (Figure 5.1). When designers create multiple distinct prototypes prior to committing to a final direction, several important benefits arise. First, alternatives provide designers with a more complete understanding of a design space [83]. Second, developing different "what if" scenarios enables more effective, efficient decision making within organizations [222]. Third, discussing multiple prototypes helps project stakeholders better communicate their requirements [157]. Finally, presenting multiple alternatives in user studies facilitates participants' ability to understand design tradeoffs and offer critical feedback [243].

Placing "enlightened trial and error" at the core of design raises the research question, *how might authoring environments support designers in creating and managing design options?* Traditionally, design tools have focused on creating single artifacts [240]. Research in subjunctive interfaces [177] pioneered techniques for parallel exploration of multiple scenarios during information exploration. Set-based interaction techniques have also been introduced for graphic design [241,242] and 3D rendering [181]. Providing alternative-aware tools for interaction design adds the challenge of working with two distinct representations:

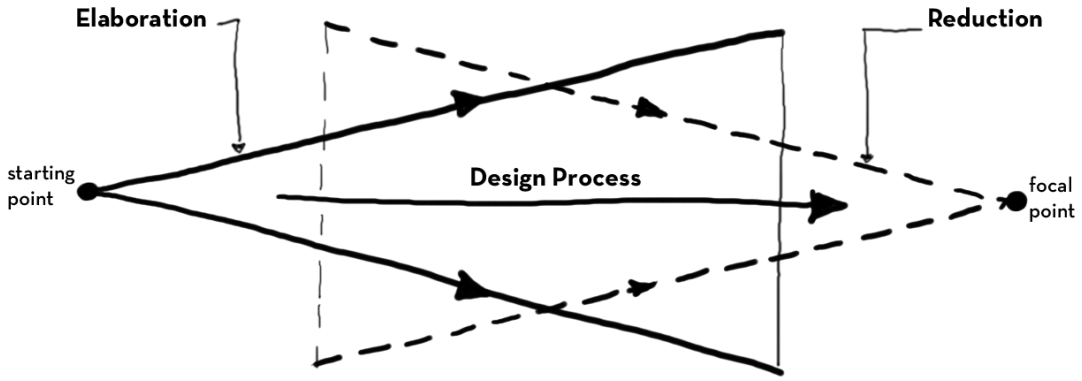

Figure 5.1: Design alternates between divergent and convergent stages. Diagram due to Buxton [55], redrawn by the author.

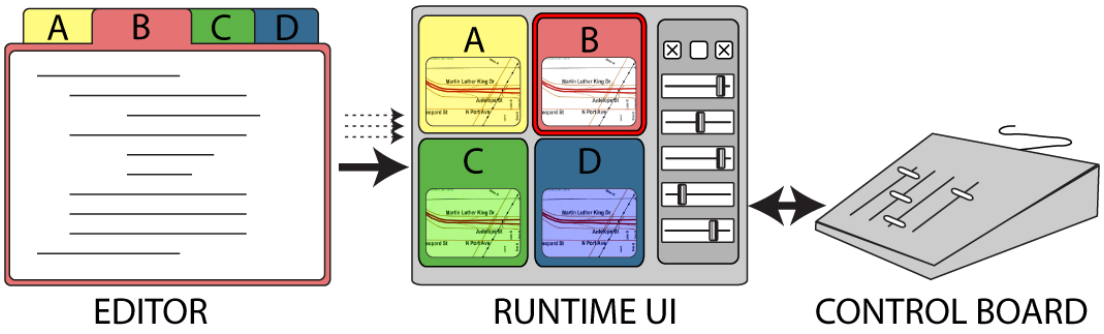

Figure 5.2: Interaction designers explore options in Juxtapose through a source code editor that supports alternative code documents (left), a runtime interface that offers parallel execution and tuning of application parameters (center), and an external controller for spatially multiplexed input (right).

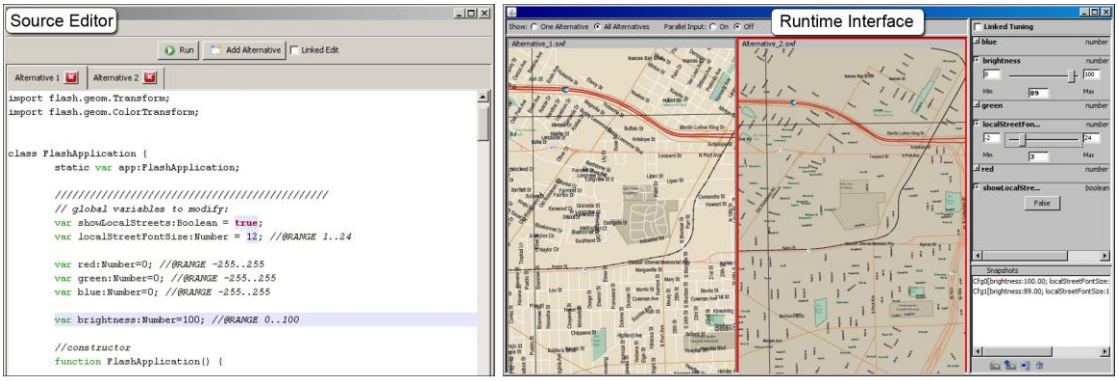

Figure 5.3: In the Juxtapose source editor (left), users work with code alternatives in tabs. Users control whether modifications affect all alternatives or just the presently active alternative through linked editing. In the runtime interface (right), alternatives are executed in parallel. Designers tune application parameters with automatically generated control widgets.

source code, where changes are authored; and the running program, where changes are observed.

This chapter suggests that interaction design tools can successfully scaffold exploration by managing alternatives across source and execution environments, and introduces Juxtapose, an authoring tool manifesting this idea (Figure 5.2). Juxtapose makes two fundamental contributions to design tool research.

First, it introduces a programming environment in which interaction designers create and run multiple program alternatives in parallel (Figure 5.3 left). Juxtapose extends linked editing [244], a technique to selectively modify source duplicates simultaneously, by turning source alternatives into a set of programs that are executed in parallel. The Juxtapose runtime environment enables interacting with these parallel alternatives.

Second, Juxtapose introduces "tuning" of interface parameters at runtime by automatically generating a control interface for application parameters through source code analysis and language reflection (Figure 5.3 right). We hypothesize that runtime controls encourage real-time improvisation and exploration of the application's parameter space. Designers can save parameter settings in presets that Juxtapose maintains across alternatives and executions. To facilitate simultaneous control over multiple tuning parameters, a physical, spatially-multiplexed control surface is supported.

This chapter first introduces findings from formative interviews that motivate our work. We then describe the key interaction techniques for creating, executing, and modifying alternatives with Juxtapose. We describe implementations for desktop, mobile, and tangible applications. Next, we present evaluation results and conclude by discussing tradeoffs and limitations of our approach.

5.2 FORMATIVE INTERVIEWS

To augment our intuitions from our own teaching and practice, we conducted three interviews with interaction designers. Here, we briefly summarize the insights gained.

First, arriving at a satisfying user experience requires *simultaneous adjustment of multiple interrelated parameters*. For example, a museum installation developer shared that getting an interactive simulation to "feel right" required time-intensive experimentation with parameter settings. Similarly, an instructor for a course on computer-vision input in HCI reported that students found adjusting recognition algorithm parameters to be a lengthy trial-and-error process.

Second, *creating alternatives of program logic* is a complementary practice to parameter tuning. In one participant's code, we saw multiple alternative code strategies living side-by-side inside a single function (Figure 5.4). To try out these different approaches in succession, this interviewee would change which alternative was uncommented (i.e., active), recompile, and execute.

Lastly, all interviewees reported writing custom control interfaces for internal program variables when they were unsure how to find good values. These tuning interfaces are not actually part of the functionality of the application — they function exclusively as exploratory development tools.

Across the three concerns, interviewees resorted to ad-hoc practices that allowed for some degree of exploration despite a lack of tool support. The following scenario illustrates

```
int calculateNextSize(int [][]currentSizes, int i, int j) {
  float denominator = 0;
  int sumOfNeighbors = 0;
  int maxOfNeighbors = 0;
  if(i != 0) {
    sumOfNeighbors += currentSizes[i - 1][j]; denominator += 1;
    maxOfNeighbors = currentSizes[i - 1][j];
//    if(j != 0) sumOfNeighbors += currentSizes[i - 1][j - 1]/2; denominator += .5;
//    if(j != currentSizes[0].length - 1) sumOfNeighbors += currentSizes[i - 1][j + 1]/2; denominator += .5;
  }
  if(i != currentSizes.length - 1) {
    sumOfNeighbors += currentSizes[i + 1][j]; denominator += 1;
    if(currentSizes[i + 1][j] > maxOfNeighbors) maxOfNeighbors = currentSizes[i + 1][j];
//    if(j != 0) sumOfNeighbors += currentSizes[i + 1][j - 1]/2; denominator += .5;
//    if(j != currentSizes[0].length - 1) sumOfNeighbors += currentSizes[i + 1][j + 1]/2; denominator += .5;
  }
```

Figure 5.4: Example code from our inquiry: two behaviors co-exist in the same function body. The participant would switch between alternatives by changing which lines were commented.

how Juxtapose can improve such exploration by explicitly addressing parameter variation, alternative creation and control interface generation.

5.3 EXPLORING OPTIONS WITH JUXTAPOSE

Tina is designing the graphical interface for a new handheld GPS device that both pedestrians and bicyclists will use. She imagines pedestrians will pan the map by tilting the device, and use buttons for zooming. Bicyclists mount the device in a fixed position on their handlebars, so they will need buttons to pan and zoom.

To try out navigation options, Tina loads her existing map prototype and clicks the Add Alternative button (Figure 5.5A); this duplicates her code in a new tab. With the Linked Edit box checked, she adds a function to respond to button input. This code change propagates to both alternatives. She clears the Linked Edit checkbox so that she can write distinct input handlers in the function body of each alternative (Figure 5.5B). In unlinked mode, edits only apply to the active tab. A colored background highlights code that differs between alternatives (Figure 5.5C).

Tina executes her designs. Juxtapose's runtime interface shows the application output of each code alternative side-by-side (Figure 5.5D). One alternative is active, indicated by a red outline. Global Number and Boolean-typed variables of this alternative are displayed in a variable panel to the right of the running applications. Tina expands the entries for layer visibility, panning speed and zoom step size to reveal tuning widgets that allow her to change values of each variable interactively (Figure 5.5E). Tina uses the tuning widgets to arrive at fluid pan and zoom animations.

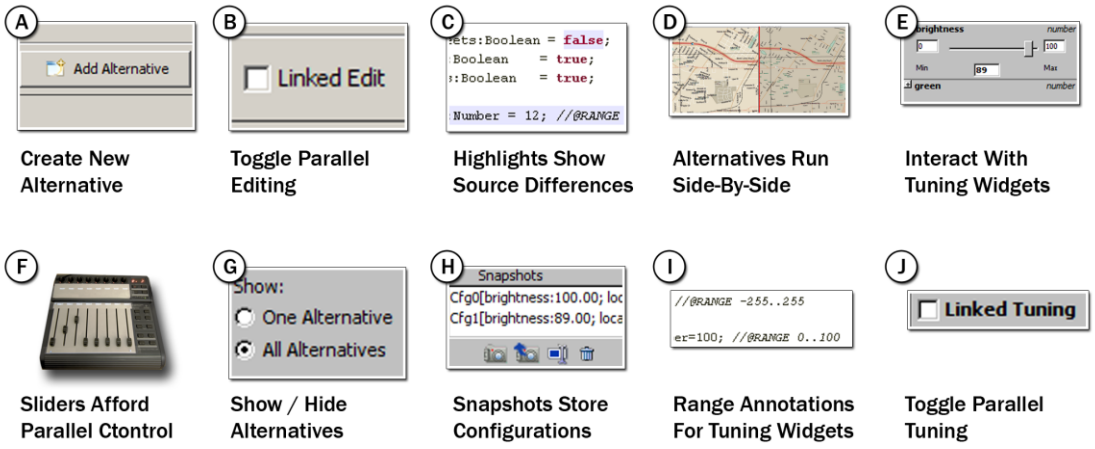

Figure 5.5: UI vignettes for the Juxtapose Scenario.

Tina also hypothesizes that bicyclists will value velocity-contingent visual and typographic levels of detail. To adjust the text sizes of multiple road types simultaneously, she moves her non-dominant hand to an external physical control board (Figure 5.5F). She places one finger on each slider, and quickly moves multiple sliders simultaneously to visually understand the gestalt design tradeoffs, such as legibility and clutter. To focus in on the details of one alternative, she toggles between viewing alternatives side-by-side, and viewing just one alternative (Figure 5.5G).

Tina finds several promising parameter combinations for showing levels of detail and uses the snapshot panel to save them (Figure 5.5H). Back in the code editor, she introduces a speed variable to simulate sensed traveling velocity, and adds code to load different snapshots from the Juxtapose environment when the speed variable changes. To constrain tuning to useful values, she adds range annotation comments, e.g., indicating that speed should vary between 1 and 30 mph (Figure 5.5I). She runs her design again and selects speed for tuning. Moving the associated slider now switches between the snapshot values she previously saved. She checks the Linked Tuning box to propagate changes in simulated speed to all alternatives in parallel (Figure 5.5J).

5.4 ARCHITECTURE FOR ALTERNATIVE DESIGN

This section outlines fundamental requirements for parallel editing, execution, and tuning, and describes how the Juxtapose implementation supports these techniques.

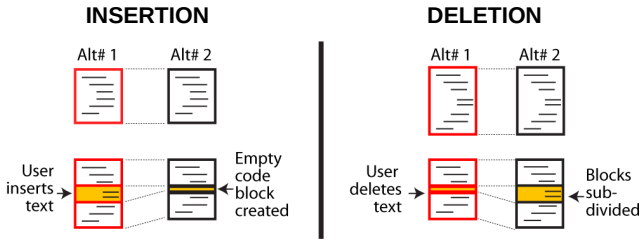

Figure 5.6: Juxtapose's implementation of linked editing is based on maintaining block correspondences between alternatives across document modifications.

5.4.1 PARALLEL EDITING

To make working with multiple code alternatives feasible, an authoring environment must keep track of code differences across alternatives, make this structure visually apparent to the user, and offer efficient interaction techniques for manipulating content across alternatives. To support these three requirements, Juxtapose extends Toomim et al.'s linked editing technique [244]: alternatives are accessible through document tabs; source differences between tabs are highlighted with a shaded background; and edits can be either local to one alternative or global to all alternatives. Toomim's work focused on sharing code snippets across different locations within a project. Juxtapose instead targets creation of sets of applications based on a core of shared code. To enable interactive editing across multiple documents, Juxtapose replaces Toomim's algorithm with incremental correspondence tracking during editing and slower content differencing during compilation. The efficiency gains thus realized enable Juxtapose to run comparisons after each key press. Average times for single character replacement operations were under 1 ms with up to 5 alternatives on a 2 GHz PC running Windows Vista.

Juxtapose tracks correspondences between alternatives by partitioning all source alternatives into corresponding blocks. In linked editing, the block structure stays fixed and block content is modified in all alternatives. In unlinked editing, code blocks are subdivided and alternatives store different content in their sub-blocks (Figure 5.6). When inserting text while unlinked, Juxtapose's data structure splits the code into pre- and post-insertion blocks and creates a new code block for the inserted text. Juxtapose splits all alternatives, inserting an empty element into the unmodified alternatives. Deletions also split code blocks. Here, the active document represents the deletion with an empty element; the corresponding elements in the other alternatives contain the deleted text. Code modifications are expressed as deletions followed by insertions. Blocks are never merged during editing.

Incremental structure tracking performs differently than content-based matching if a user types identical code into corresponding locations in two distinct documents: content-based approaches will mark this as a match; structure-based approaches will not. To obtain both interactive performance and content matching, Juxtapose optimizes global block structure with a slower longest common subsequence algorithm at convenient times (i.e., when compilation is started).

5.4.2 PARALLEL EXECUTION AND TUNING

Executing a set of related interaction designs raises two principal questions: Should alternatives be presented in series or in parallel? And should users interact with these alternatives one-at-a-time or simultaneously? To investigate how different target devices offer unique opportunities for parallel input and output, we implemented versions of the Juxtapose environment for three domains: desktop interactions written in ActionScript for Adobe Flash; mobile phone interactions for Flash Lite; and physical interactions based on the Arduino microcontroller platform. The three implementations share a common editor but differ in their runtime environment. We discuss each in turn.

DESKTOP

Desktop PCs offer sufficient screen resolution to run alternative interactions side-by-side, analogous to application windows. In our implementation, alternatives are authored in ActionScript 2, from which Juxtapose generates a set of Flash movie files using the MTASC compiler [27]. The generated files are then embedded into the Juxtapose Java runtime interface using a Windows-native wrapper library [28]. For consistency with the temporally multiplexed input of windowed operating systems, only one active alternative receives keyboard and mouse input events by default. However, Juxtapose offers the option to replicate user input across alternatives through event echoing [176]. By using a provided custom mouse class, mouse events can be intercepted in the active alternative and injected into all other alternatives, which then show a ghost cursor. This parallelism only operates at the low level of mouse move and click events, which is useful when both application logic and visual layout are similar across alternatives. However, in the absence of a model that translates abstract events in one application into equivalent events in another, users cannot usefully interact with different application logic simultaneously. While development of an abstract input model that provides such a mapping is certainly possible, it is unlikely to occur during prototyping, when the application specification is still largely in flux.

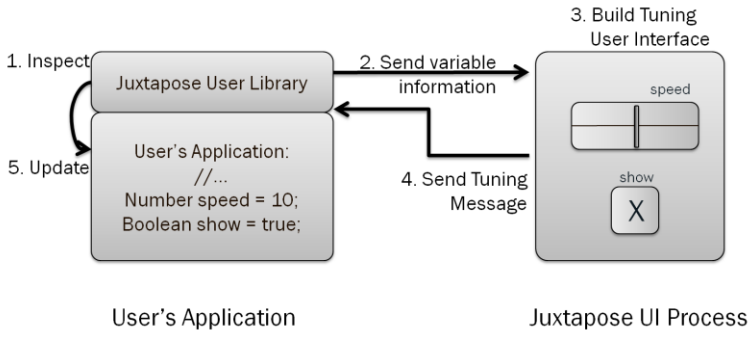

Figure 5.7: Runtime tuning is achieved through bi-directional communication between a library added to the user's application and the Juxtapose runtime user interface.

To accomplish runtime variable tuning, bi-directional data exchange between the user's application and the tuning interface is required. On startup, the application transmits variable names, types, and values to Juxtapose (Figure 5.7). The tuning interface in turn sends value updates for variables to the application whenever its widgets are used. Loading snapshots defined in the tuning interface from code is initiated by a request from the user application, followed by a response from Juxtapose. To accomplish this communication, the user adds a Juxtapose library module to their code. In our implementation, communication between the Flash application and the hosting Java environment takes place through a message-passing protocol and synchronous remote procedure call interface built on top of the Flash Player API.

MOBILE PHONE

For smart phones, the most useful unit of abstraction for parallel execution might not be an application window on a handset, but rather the entire handset itself. The small form factor and comparatively lower cost make it attractive to leverage multiple physical devices in parallel (Figure 5.8). In Juxtapose mobile, developers still compose and compile applications on a PC. At runtime, the tuning interface resides on the PC, and the alternatives run on different handsets. A designer can rapidly switch between alternatives by putting one phone down and picking another one up. To target tuning events to an application running on a particular phone, Juxtapose offers alternative selection buttons in the runtime interface.

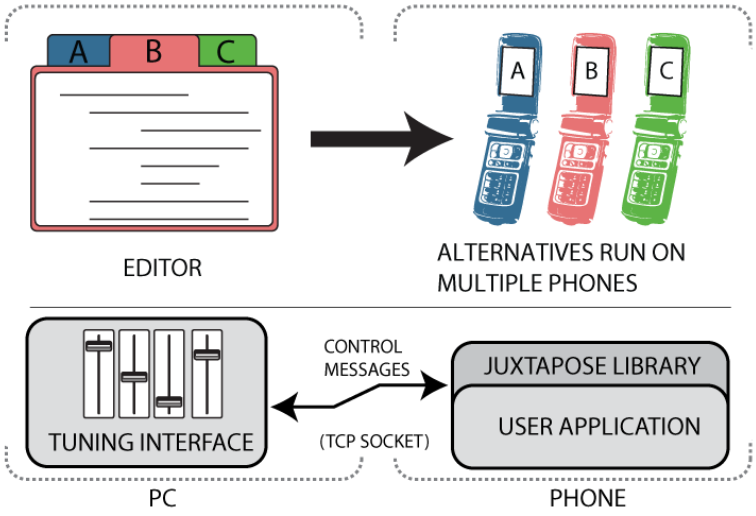

Figure 5.8: When using Juxtapose mobile, code alternatives are executed on different phones in parallel. Variable tuning is accomplished through wireless communication.

Our Juxtapose mobile prototype generates binaries which run on the Flash Lite 2.0 player on Nokia N93 smart phones. The desktop tuning interface and the smart phone communicate through network sockets. When designers run an application on the mobile phone, it opens a persistent TCP socket connection to the Juxtapose runtime interface on the PC. Our prototype uses Wi-Fi for simplicity. Informally, we found that the phone receives variable updates at approximately 5 Hz, much slower than on the PC, but still sufficient for interactive tuning. Response rates are slower because mobile devices trade off increased battery life for slower network throughput and increased latency. A limitation of the current

Figure 5.9: Two prototypes built with Juxtapose mobile. Left: A map navigation application explored use of variable tuning. Right: Two alternatives of a fisheye menu navigation technique running on two separate phones.

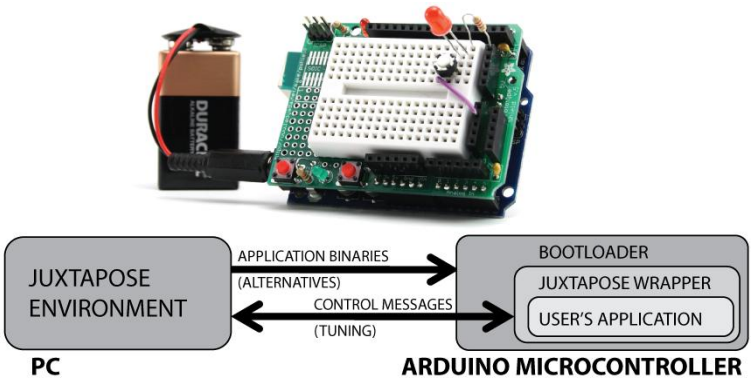

Figure 5.10: For microcontroller applications, Juxtapose transparently swaps out binary alternatives using a bootloader. Tuning is accomplished through code wrapping.

Juxtapose mobile implementation is that users must manually upload compiled files to the phones and launch them within the Flash Lite player. This is due to restrictions of the phone's security architecture. We have explored the utility of Juxtapose mobile with several UI prototypes, including map navigation and fisheye menus (Figure 5.9). While the latency of tuning messages made the external MIDI controller less useful in our tests (it generates too many events which queue up over time), the ability to modify the application running on the phone while another user is interacting with that phone appeared to be especially useful.

PHYSICAL INTERACTIONS

Many interaction designers work with microcontrollers when developing new physical interfaces because they offer access to sensors and actuators. The primary difference to both desktop and mobile development is that novel physical interaction design involves building custom hardware, which is resource intensive. Consequently, designers are likely to embed multiple different opportunities for interaction into the same physical prototype.

Juxtapose supports developing for the Arduino [185] platform and language, a combination popular with interaction designers and artists. Code for all alternatives is cross-compiled with the AVR-GCC compiler suite. Juxtapose for Arduino uploads and runs only one code alternative on one attached Arduino board at a time. When the designer switches between alternatives, Juxtapose transparently replaces the binary running on the microcontroller through a bootloader (Figure 5.10).

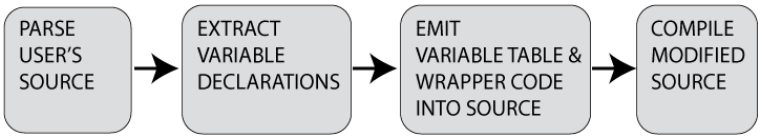

Figure 5.11: The pre-compilation processing step extracts variable declarations and emits them back into source code as a symbol table.

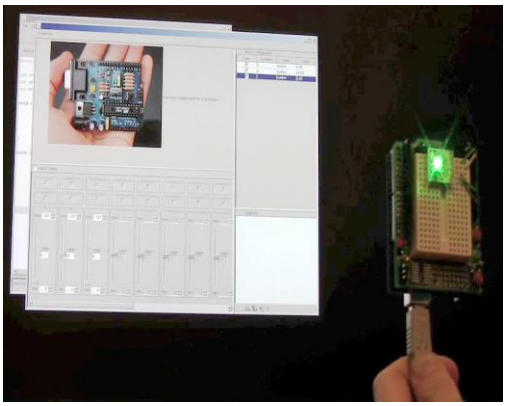

Figure 5.12: Example application demonstrating live tuning of color parameters of a smart multicolor LED through the Juxtapose runtime user interface.

Real-time tuning of variables requires a mapping from variable names to types and storage locations, which is not available in the C language that Arduino uses. Juxtapose constructs this map using a preprocessing step that transforms a user's program before compilation (Figure 5.11). The user's source code is parsed to build a table of global variable names, types, and pointers to their memory locations. The source is then wrapped in Juxtapose-specific initialization code, into which the variable table is emitted as C code. When a variable is tuned (Figure 5.12), the embedded wrapper code uses this table to find a pointer to the correct runtime variable from its name and changes the value of the memory location. The wrapper code also contains communication functions to exchange information between microcontroller and PC through a serial port. Some price must be paid for this added flexibility. The developer has to relinquish control of a hardware serial port, and application state is lost whenever alternatives are switched. Snapshots provide a way to save and restore values across such changes.

5.4.3 WRITING TUNABLE CODE

Ideally, programmers should be able to leverage tuning and alternatives in their project without changing their source. In practice, tuning is invisible unless modified parameter values have some observable effect on program execution. In other words, the changed variable has to be read again and some action has to be taken based on its value after it was modified at runtime. Thus programmers may have to write additional code that is solely concerned with making their application tunable.

To help programmers express the logic for runtime updates, callback functions provide a lightweight harness: whenever a variable is tuned at runtime, the application is notified of the parameter name and its updated value. In ActionScript, this callback facility is already provided on the language level by the `Object.watch()` method. The following example calls a redraw routine whenever the variable tunable is updated by the Juxtapose tuning UI:

```
01  var tunable = 5; //@RANGE 0..100
02  var counter; //@IGNORE
03  var callback= function(varName,oldVal,newVal){
04    redraw();
05    return newVal;
06  }
07  this.watch('tunable',callback);
```

Beyond callbacks, protocols to communicate information from the source code to the runtime interface enable designers to initialize the runtime UI programmatically. Programmers can specify minimum and maximum values for Number variables through comment annotations (line 1). They can also hide variables for which tuning is not useful, e.g., counters, from the variable list (line 2). Code annotations have been used in other projects as a source of meta-information, e.g., for labeling different experimental conditions for user testing [180]. Juxtapose currently uses code comments to capture annotations; this functionality could become part of the language definition in an alternative-aware programming language.

5.4.3.1 Hardware Support

Three important benefits can be realized by using a dedicated external controller instead of mouse and keyboard input for parameter control. First, spatially multiplexed input enables users to modify multiple parameters simultaneously. Second, with mouse control, tuning is mainly a hand-eye coordination task — with a dedicated control board, it turns into a motor task that leaves the eyes free to focus on the application being tuned. Third, moving the tuning UI to a dedicated controller allows for tuning of interactions that require mouse and

Figure 5.13: An external controller enables rapid surveying of multidimensional spaces. Variables names are projected on top of assigned controls to facilitate mapping.

keyboard input, e.g., adjusting the rate at which mouse wheel movement magnifies a document.

Our implementation supports a commercially available USB MIDI device [29] with 16 buttons with LED status indicators, 8 rotary encoders (presently not used) and 8 motorized faders (Figure 5.13). The controller transmits input events as MIDI control change messages and receives similar control change messages to actuate sliders and toggle LED feedback. Actuation of the hardware controller is essential for saving and restoring parameter snapshots — without actuation it is impossible to recall saved parameter values and edit them incrementally. To facilitate locating a particular variable's control, the mixer was augmented with a small top-mounted projector which displays parameter names next to the appropriate controls, a technique inspired by Crider et al. [65]. While a projector setup is unwieldy in practice, controllers with embedded text LCDs that can offer the same functionality are commercially available.

5.5 USER EXPERIENCES WITH JUXTAPOSE

To evaluate the authoring approach embodied in Juxtapose, we built example prototypes using the tool and conducted a summary usability study of Juxtapose for desktop applications. We recruited 18 participants, twelve male, six female. Participants were undergraduate and graduate students with HCI experience. Their ages ranged from 20 to 32

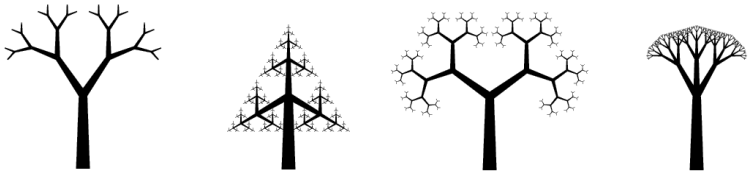

Figure 5.14: Study participants were given a code example that generates images of trees. They were asked to then match the four tree images shown above.

years. All but one participant had at least working knowledge of procedural programming and all had at least some expertise in interaction design.

5.5.1 METHOD

Evaluation sessions lasted approximately 75 minutes. Participants were seated at a workstation with mouse, keyboard and MIDI controller. After a demonstration of Juxtapose, participants were given three tasks. The first task was a warm-up exercise to modify a grid animation reacting to mouse movement, adapted from the book Flash Math Creativity [206]. Participants were asked to make changes that required both code alternatives and tuning.

The second task was a within-subject comparison that asked participants to adjust four parameters of a recursive tree-drawing routine to match four specific tree shapes (Figure 5.14). The provided code was also adapted from Flash Math. For two trees, this was accomplished using the full Juxtapose interface. For the other two, participants were given the same editor without the possibility of creating alternatives or tuning. Order of assignment between Juxtapose and control conditions was counterbalanced and a random tree order was generated for each participant.

The third task asked participants to work on the mapping scenario introduced earlier. They were provided with a working ActionScript program that loaded a map containing 28 different layers of information (e.g., land areas, parks, local streets, local street names, highways). Participants were given 30 minutes to create two map navigation alternatives. They were then asked to present their maps to a researcher. Documentation contained examples for how to programmatically change visibility of layers, color and brightness, text size and formatting, and mouse interactions. Participants had to modify and add to these examples to either hardcode design decisions or to set up tunable parameters through callback functions in the source code.

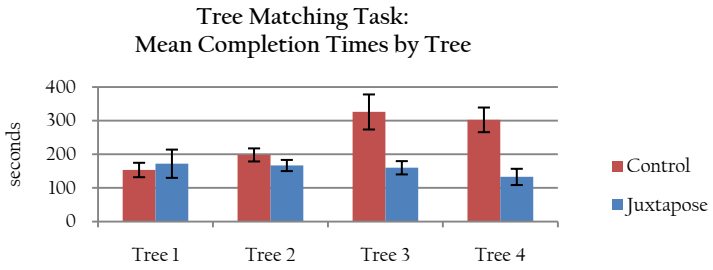

Figure 5.15: Study participants were faster in completing the tree matching task with Juxtapose than without.

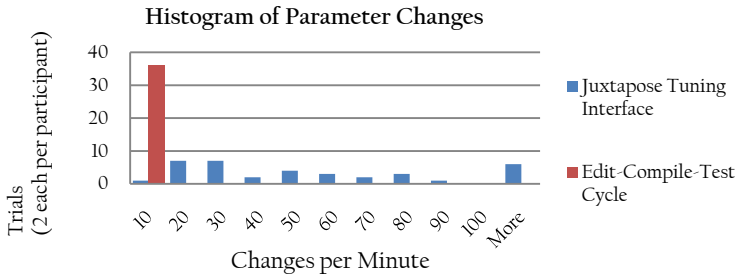

Figure 5.16: Study participants performed many more design parameter changes per minute with Juxtapose than without.

5.5.2 RESULTS

In all tasks, all participants properly applied linked and unlinked editing and tuning, with no apparent confusion. Participants commented positively on the ease of adjusting numerical parameters through tuning and the reduced iteration time this permitted. One participant commented that the explicit management of alternative documents improved on their existing practice of "half-hearted attempts to name saved [configurations] with memorable names." Today, designers commonly use layer sets as a technique for composing alternatives in graphics. A participant commented that Juxtapose brings this pattern to interaction design.

TUNING ENABLES MORE PARAMETER EXPERIMENTATION, FASTER

In the tree matching task, participants took an average of 258 seconds (σ: 133 s) to complete the matching in the control condition, and an average of 161 seconds (σ: 82 s) to complete the task with Juxtapose. This difference was significant (one-tailed, paired Student's t-test; p ‹

0.01). When looking at completion times by tree (Figure 5.15), a large discrepancy for trees three and four becomes apparent. For these trees, participants quickly narrowed in on the approximate shape but frequently had trouble minimizing the remaining visual disparity when they could no longer reason about how to proceed toward the goal. Participants then often broadened their search in parameter space and diverged from the solution while looking for the right parameters to adjust. We believe that Juxtapose outperformed the control condition here because the penalty for an uncertain, diverging move was much smaller — the result could immediately be observed and corrected.

To quantify the cost of making a change, we investigated how many parameter combinations participants explored. In the control condition, on average, participants tested 2.60 parameter combinations per minute to arrive at matches (σ : 0.93; we counted each execution after changing source as one combination). In contrast, using Juxtapose, participants executed the Flash file only once, and generated parameter changes through the tuning interface. Here participants explored 64 combinations on average (σ : 80; we counted each variable change sent to Flash as a tuning event). The external MIDI controller generated many input events and one might contend that our definition of parameter change over-estimates the number of perceptually different states explored by users. We note that participants adopted a wide range of tuning strategies — some exclusively typing in numbers in the tuning interface, others using multiple sliders simultaneously. This resulted in a wide spread of parameter changes per minute for Juxtapose (Figure 5.16), but even participants at the lower end of the histogram explored an order of magnitude more states than participants in the control condition.

ALTERNATIVES & TUNING PROVIDE VALUE, AT A PRICE

In our mapping task, many participants began by adding instrumentation code to the provided framework to make map attributes tunable at runtime. While hard-coding design choices into source code would have been easier from a programming perspective, participants spent extra effort to make variables tunable so they could experiment at runtime. Two participants mixed strategies, making some parameters tunable while setting others in code in different alternatives when they were sure about their desired values. For example, one participant hard-coded a higher initial magnification factor in the pedestrian map interface.

Most participants preferred to set the ranges for Number variables in source code, not in the runtime interface. Only one participant used the runtime interface for this purpose. A

possible explanation is that reasoning about ranges has to do with how a variable is used in the source so participants were more inclined to express ranges there.

SUGGESTIONS FOR IMPROVEMENT

The map task also uncovered a number of usability shortcomings. In multiple instances, participants closed the runtime window to change a line of code and recompile, discovering that their runtime parameter settings from the last execution were gone. To address this, Juxtapose could automatically save the last parameter values in a snapshot when the runtime window is closed.

Participants also wished for a larger range of variables to access — for the study, only variables declared in the main application class and variables of the root object of the visual hierarchy were accessible for tuning. Participants thus had to introduce intermediate variables to influence other graphical objects. It would be preferable to have a "tuning mode" for direct manipulation of all graphical objects, extending ideas introduced in SUIT [203].

Many participants expressed frustration at the lack of search and undo in the source editor. Both could clearly be added. Multiple participants also felt that it was overly onerous to properly write the application callbacks that make a design tunable. This can be addressed in two ways. Directly modifying object fields can be handled by making all fields tunable, not just global variables. More complex parameter mappings however will still require callbacks: producing these callbacks can be supported through a code generation wizard.

5.6 LIMITATIONS & EXTENSIONS

Juxtapose focused on exploring alternatives of user interfaces that were programmatically defined within a single file of source code. The design choices made during the development of Juxtapose represent one particular point in a larger space of tools for explorative programming. In this section, we discuss assumptions made in our current design and highlight limitations of our implementation. Following Fitzmaurice's design space for graspable interfaces [78], we summarize the most salient design decisions in Figure 5.17. This design space is not meant to be exhaustive — it covers the decision points encountered during prototyping and development. Nevertheless, the table suggests additional techniques, such as automatic generation of alternatives, which may be a fruitful area for future work.

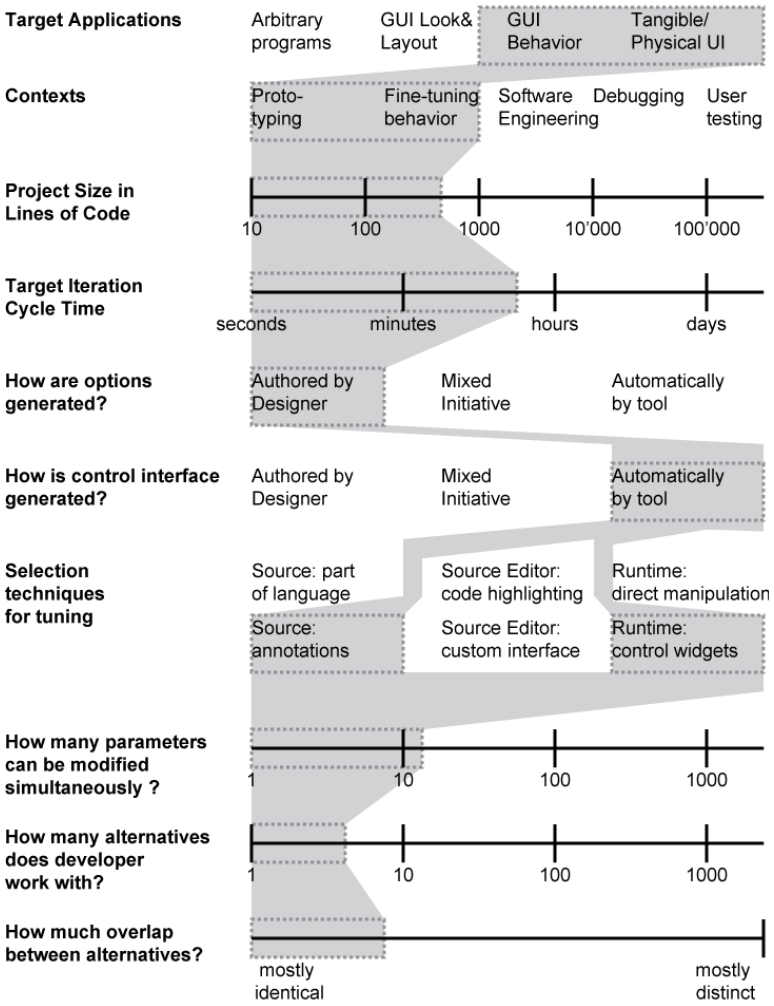

Figure 5.17: A design space for exploring program alternatives. Choices implemented by Juxtapose are shown with a shaded background.

5.6.1 WILL DESIGNERS REALLY BENEFIT FROM LINKED SOURCES?

The efficacy of linked editing in Juxtapose rests on the assumption that interaction designers create multiple alternatives of a common code document, where individual alternatives only differ in parameter settings and small sections of code. Experimenting with code in this manner only covers part of the solution space for a given problem. Different solution approaches may be based on distinct implementations. Alternatives as discussed in this paper explore options within one particular solution strategy. Are alternative designs related enough in practice to benefit from linked editing and tuning?

Beyond evidence from our formative interviews, the book Flash Math Creativity [206] provides detailed examples of source code experimentation by professionals: 15 Flash designers share how they create computational designs in 56 projects. Each project starts from a single idea, e.g., animating geometric grid structures. The designers then show how they modified the initial source to explore the design space. 12 of 15 designers showed multiple alternatives for their projects (mean: 10.2 alternatives per project; range: 3 to 23). The difference between these alternatives is usually small: a change to a line of code to load different graphics, alterations to parameter values, or substitutions of function calls.

5.6.2 IS TUNING OF NUMBERS AND BOOLEANS SUFFICIENT?

Juxtapose's runtime tuning focuses on direct manipulation of Boolean and Number types. Would designers benefit from more expressive abstractions and additional functionality in the tuning interface?

An underlying assumption in this work is that developers both produce the application and tune it. If they desire a more complex mapping, e.g., a logarithmic parameter scale, they may express this mapping in the source. Locating additional functionality in the source itself may be more useful since logic expressed in the tuning UI is not available when the application is run outside Juxtapose. This assessment changes if alternatives and tuning options are used by a third party, e.g., during participatory design sessions. In this case it would make sense to imbue the runtime interface with more flexibility to let users express a more complete set of modifications without editing the program source, e.g., by providing rich widgets for commonly used complex data types such as colors or coordinates.

5.6.3 ARE CODE ALTERNATIVES ENOUGH?

Perhaps the most important limitation is that Juxtapose does not offer support for managing multiple alternatives of graphical assets. Interface design is concerned with both look and feel — graphics and behavior. Many popular user interface authoring tools today follow a hybrid authoring approach, where graphical appearance is edited through visual direct manipulation, while behavior is specified in source code (e.g., Flash [1], Director [6]). We believe Juxtapose is a first step towards an integrated authoring environment that offers management of alternatives across graphics and code. Future research should investigate to what extent it is possible to offer a coherent method of exploring alternatives for both, in a single tool. The most relevant prior work for exploring graphical alternatives is Terry's work on embedding alternatives for graphics manipulations into a single canvas [242], and research on editable

graphical histories [153,236]. However, a naive crossproduct of Juxtapose's linked editing and graphical alternative or history techniques is unlikely to work, because it would likely overburden the user with too many inconsistent methods of making choices. The goal of future research should be to find a single, "simple-enough" mental model.

5.6.4 ALTERNATIVES FOR COMPLEX CODE BASES

Another open question is how an alternative-aware editor could be extended to handle large software projects. Juxtapose targeted UI prototypes, for which interaction logic is frequently authored in a single source file today. If the goal is not the design of a new UI, but the augmentation of an existing program, designers may have to contend with large existing code bases. For example, a software engineer at Adobe reported that to try alternatives for a new feature in a large authoring tool, he would have to check out several thousand files into independent workspaces, and manage any changes between alternatives manually [94].

As an interaction technique, we have envisioned the use of hierarchical tabs where the top level identifies the alternative, and a lower level identifies the file within the alternative.The primary challenge will be to reduce the potential complexity stemming from dealing with multiple alternatives in the authoring interface. As an implementation strategy, it would be interesting to consider to what extent virtualization technology can be harnessed to quickly create independent copies of complex applications and system configurations that are adequately isolated from each other.

5.6.5 SUPPORT EXPLORATION AT THE LANGUAGE LEVEL

Juxtapose chose to implement support for runtime tuning at the library level — the source language, ActionScript in the case of Juxtapose, remained unchanged. Juxtapose shares this approach with prior work like Amulet [190]. Operating as a library has the advantage that Juxtapose can target a widely used language; it has the drawback that the program has to be explicitly changed to include library support. More importantly, the library has limited control over program execution at runtime. For example, when running multiple alternatives side by side, it is not possible to pause execution of one application as it loses focus — all applications run in parallel, even if interaction with them is sequential.There are two possible ways for future research to extend the reach of runtime exploration:

1) augment an existing programming language with additional language constructs
2) develop a new language to provide explicit developer control over alternatives and variable parameter spaces.

Terry's Partials project [239: Appendix B] was an exploration of the first option. He augmented the Java language with the keywoard "partial" which could be used to decorate variable definitions to gain runtime control over those variable values. It is worthwile to explore what benefits an entirely new language targeted at exploration could provide.

5.6.6 INTEGRATE WITH TESTING

A final direction worth pursuing in future work is to extend parallel editing and tuning to support user testing of alternatives. A particularly promising application domain would be the authoring of user interfaces for web applications, since online deployment could provide a way to rapidly gather empirical data on user preferences for different alternatives. Large web sites already routinely test alternatives of new features by running controlled bucket experiments: a small percentage of site visitors are exposed to a new proposed feature or layout, and results (time spent on site, purchases made) are compared with the control condition [16]. An interesting an as-of-yet unexplored research question is to what extent such comparative testing with remote users is possible during earlier prototyping stages.

5.7 SUPPORTING ALTERNATIVES IN VISUAL PROGRAMS

How might support for alternative behavior transfer from the textual programming domain of Juxtapose into visual authoring environments such as d.tools? Following our implementation of Juxtapose, we examined to what extent the advantages of defining and editing multiple alternatives can be realized within d.tools. We have not yet investigated how to transfer variable tuning; partially because variables play a less prominent role within d.tools projects. Because d.tools focuses on user interfaces with custom hardware, parallel execution of alternatives is less likely to be useful. We therefore focused on expressing and managing alternatives in the editor, but only support executing one alternative at a time.

What level of abstraction should alternatives operate on? Juxtapose manages alternatives at the file level. For visual diagrams, this choice is also possible, but less compelling. A prototype implementation of file alternatives in d.tools suggested that making sense of the differences between alternative files is harder for visual programs than for textual ones. Specifically, changes in the visual gestalt of the diagram are not necessarily related to changes in the functionality expressed by the diagram. Rearranging states in a d.tools diagram changes appearance but not logic. We therefore sought ways to express alternatives within a single diagram, at the state level.

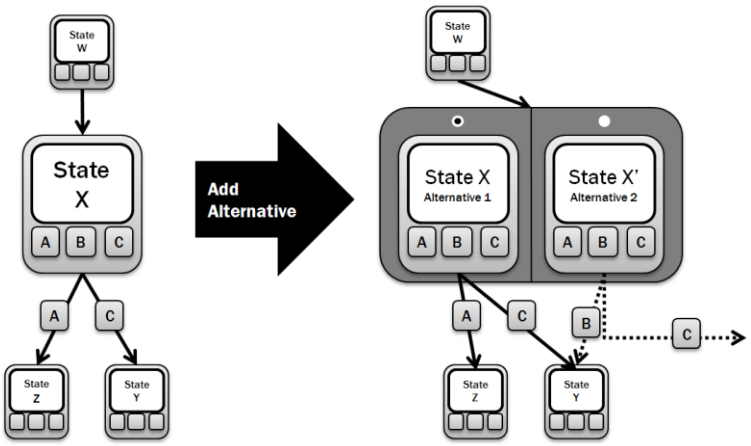

Figure 5.18: Schematic of state alternatives in d.tools: alternatives are encapsulated in a common container. One alternative is active at a time. Alternatives have different output and different outgoing transitions.

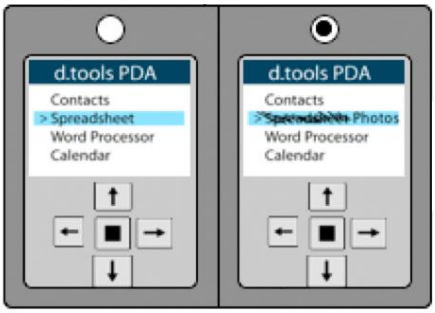

Figure 5.19: Screenshot of a d.tools container with two state alternatives. In the right alternative, screen graphics have been revised.

Designers can introduce *state alternatives* in d.tools to define both appearance and application logic. An alternative container (Figure 5.18, Figure 5.19) encapsulates two or more states. State alternatives are created in a manner analogous to the Juxtapose editor: designers select a state and choose "Add Alternative" from its right-click context menu. The original state (with all defined output such as screen graphics) is duplicated and both states are placed into an alternative contained. To express that the incoming transitions remain the same, regardless of which alternative is active, the original state's incoming connections are rerouted to point to the encapsulating container. To define which of the alternative states should become active when control transfers to an alternative container, the container shows

radio buttons, one above each contained state. Outgoing transitions are not shared between alternatives: each state can thus define its own set of target states and transition events. To reduce visual clutter, only outgoing transitions of the active alternative are shown; other outgoing transitions are hidden until that state is activated.

State alternatives support more localized changes than Juxtapose's code alternatives. If alternatives are defined for more than one state, managing correspondences between the different alternatives is currently cumbersome. Support to combine different alternatives into coherent alternative sets is needed and should be addressed in future work. State alternatives have been evaluated in laboratory studies as part of the d.note project on revising d.tools diagrams, which will be described in the next chapter.