Denicek: Computational Substrate for Document-Oriented End-User Programming

Anonymous Author(s)

Abstract

User-centric programming research gave rise to a wide range of compelling programming experiences, including local-first collaborative editing, programming by demonstration, incremental recomputation, schema change control, end-user debugging and concrete programming. Those experiences advance the state of the art of end-user programming, but they are hard to implement on the basis of established programming language and system paradigms.

We contribute Denicek, a computational substrate that reduces the complexity of implementing the aforementioned programming experiences. Denicek represents a program as a series of edits that construct and transform a document consisting of data and formulas. Denicek provides three primitive operations on series of edits, application of edits, merging of histories and conflict checking. We show that the three operations form the backbone of a direct implementation of a range of programming experiences.

We discuss the architecture of Denicek, document key design considerations and elaborate the implementation of the programming experiences listed above. To evaluate the proposed architecture, we use Denicek as the basis of a simple end-user data exploration environment. The case study shows that the Denicek computational substrate provides an appealing basis for the design of rich, interactive end-user programming systems.

Keywords

Programming Systems, Computational Substrate, End-User Programming, Programming by Demonstration, Local-First Software

ACM Reference Format:
Anonymous Author(s). 2025. Denicek: Computational Substrate for Document-Oriented End-User Programming. In *Proceedings of (Conference acronym 'XX).* ACM, New York, NY, USA, ?? pages. https://doi.org/XXXX.XXXX

1 Introduction

A computational substrate defines the basic structures using which programs are constructed, how the progam state is represented and how it is changed [?]. The choice of a substrate affects what programming experiences can be readily supported by a system. For example, object-oriented programming has been historically linked to graphical user interfaces [?], while using lists as the basic substrate enabled Lisp to become a "language laboratory" [?].

In principle, any programming experience can be developed on top of any computational substrate. However, a suitable program representation and operations for working with programs provided by the substrate can eliminate much of the complexity of implementing interesting programming experiences. For example, the reflective capabilities of Smalltalk make it easy to provide rich debugging tools [?] that are difficult to provide for C/C++ [? ?].

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
2025. ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXX.XXXX

Programming Experiences. What would be the ideal programming substrate for rich, interactive end-user programming systems? In this paper, we aim to design a programming substrate that makes it easy to develop a range of programming experiences [?]:

- *Local-First Collaboration.* Multiple users should be able to concurrently modify a single document and merge their changes, preferrably without requiring a central server [? ?].
- *Programming by Demonstration.* Allow users to construct simple programs by demonstrating the steps of the expected behavior using concrete examples [? ?].
- *Incremental Recomputation.* When a part of a document changes, formulas whose result depends on it are invalidated and, possibly, automatically recomputed [? ? ?].
- *Schema Change Control.* When the user evolves the structure of the document, affected data and formulas should automatically co-evolve to match the new structure [? ?].
- *End-User Debugging.* The user should be able to ask provenance questions [?] – why a computation resulted in a particular value and what inputs contributed to the result [?].
- *Concrete Programming.* It should be possible to reuse parts of program logic, or formulas, without introducing abstractions, that is, program against concrete values [? ?].

Two-Stage Methodology. Our approach is captured by the interior mode [?] of design science research. To *design* Denicek, we identify a number of *formative examples* – simple programming tasks that manifest one or more of the desired programming experiences (§??). Using those examples, we co-design the Denicek substrate and Webnicek (§3), a simple web-based end-user programming environment based on the substrate. Although Webnicek can be used to complete end-user programming tasks, it is optimized for testing the underlying substrate rather than for usability.

To *evaluate* Denicek, we use the substrate as the basis of Datnicek (§??), an end-user data exploration environment inspired by existing systems [? ?]. We use the second stage to evaluate suitability of the Denicek substrate for the development of end-user programming systems and report the results of our evaluation (§??).

Substrate. The Denicek substrate brings together two central design ideas. First, it represents programs as document trees consisting of nodes that can represent data, formulas, evaluated results, as well as static content. Second, Denicek does not store the document tree itself, but instead, it maintains a sequence of edit operations through which the tree was constructed and transformed.

The substrate then provides three primitive operations for working with sequences of edits. First, it can apply a series of edits to reconstruct a document. Second, it can merge two diverging edit histories. Finally, it can detect conflicts when merging histories and, for example, remove conflicting edits from one branch.

The key insight that we present in this paper is that many of the programming experiences can be obtained by leveraging the uniform document representation alongside with a suitable combination of the three primitive operations.

Changing data and formulas is done using the same primitive edit operations to manipulate the document structure. A user-interface may provide a specialized editor, but still trigger the pritive edits behind the scenes. However, interacting with elements in the document, such as a entering text in a textbox, also generates a document edit that can be merged or checked for conflicts (§??).

As we will see, past edits that demonstrate an operation done to the document can be recorded, allowing programming by demonstration (§??). Replaying such recorded edits is implemented using the merging operation, which means that recorded operations continue working even if the document structure later changes. Moreover, structural changes to the document can be done independently and then merged (§??). Evaluation of formulas also generates document edits (§??). If the evaluated edits conflict with manual edits done later by the user, the evaluated edits are removed, implementing an incremental recomputation (§??).

Contributions. The structure and contributions of this paper are:

- We present the Denicek substrate (§4) and provide a detailed description of its document representation, edit operations and the key three operations.
- We explain how a wide range of end-user programming experiences can be readily implemented using the substrate (§??) in the context of a simple web-based prototype (Webnicek).
- We document important design decisions and alternatives (§??). The analysis shows that the desired functionality requires a careful choice among interconnected design options.
- We develop an end-user data exploration system (Datnicek) on top of Denicek (§??) and evaluate the extent to which the Denicek substrate enables such development (§??).

To enable others build on top of Denicek, we share our compact and documented source code at: https://github.com/d3sprog/denicek

2 Background

What is this about? end-user programming [?] but also programming systems [?], Live [?], Interactive etc. - perhaps for a lack of better term use end-user (also notebooks)

explain computational substrate Substrate defined by [?]

citizen programmers? Maybe do not talk about "programs" because this is more documents with formulas - not that fancy programs. Computational documents?

programming by demonstration [? ?] [?]

local first [? ?]

provenance/debugging [? ? ?] [? ?] [?]

end-user debugging [?]

live/incremental recomputation [? ? ?].

visualizing results of execution projection boxes [?]

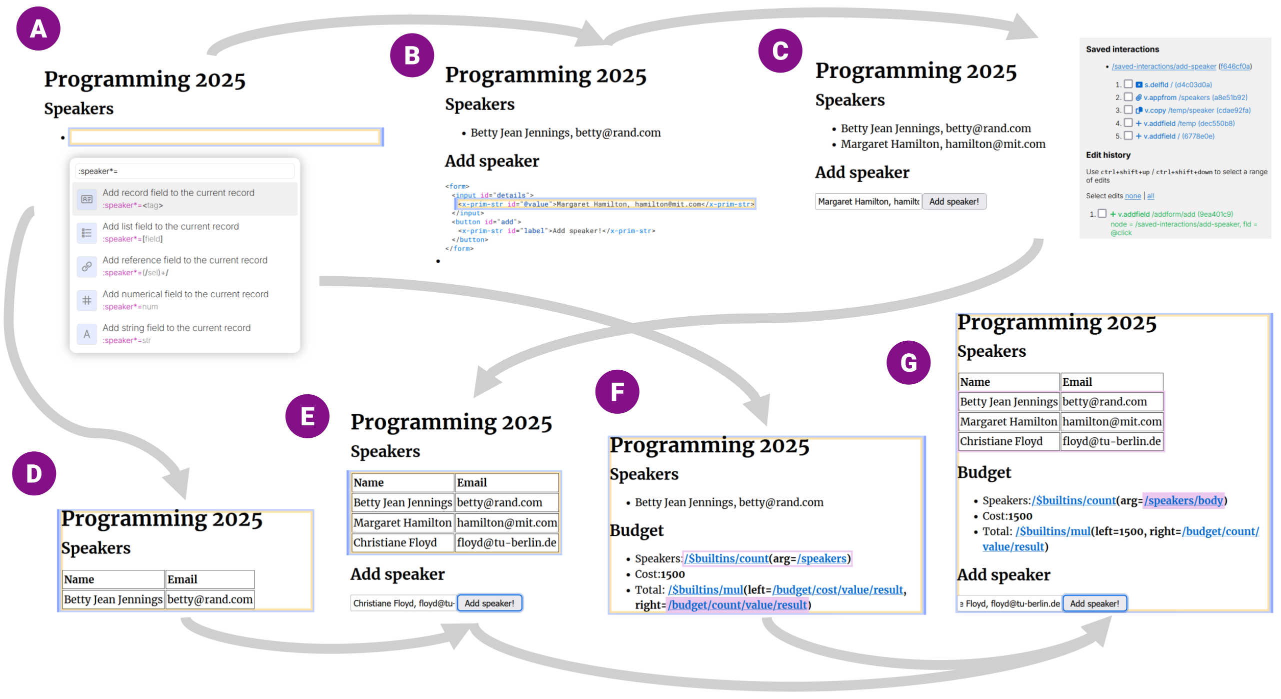
livelits (provide editors) [?]

edit calculus (based on tree navigation though) [?]

structure editing see [?]

Schema evolution

Also on substrate see



A

Programming 2025

Speakers

•

:speaker*=

Add record field to the current record
:speaker*=<tag>

Add list field to the current record
:speaker*=[field]

Add reference field to the current record
:speaker*=(/sel)+/

Add numerical field to the current record
:speaker*=num

Add string field to the current record
:speaker*=str

B

Programming 2025

Speakers

- Betty Jean Jennings, betty@rand.com

Add speaker

```
<form>
  <input id="details">
   <x-prim-str id="@value">Margaret Hamilton, hamilton@mit.com</x-prim-str>
  </input>
  <button id="add">
   <x-prim-str id="label">Add speaker!</x-prim-str>
  </button>
</form>
```

•

C

Programming 2025

Speakers

- Betty Jean Jennings, betty@rand.com
- Margaret Hamilton, hamilton@mit.com

Add speaker

Margaret Hamilton, hamilto | Add speaker!

Saved interactions

- /saved-interactions/add-speaker (f646cf0a)
  1. ☐ s.delfld / (d4c03d0a)
  2. ☐ v.appfrom /speakers (a8e51b92)
  3. ☐ v.copy /temp/speaker (cdae92fa)
  4. ☐ + v.addfield /temp (dec550b8)
  5. ☐ + v.addfield / (6778e0e)

Edit history

Use `ctrl+shift+up` / `ctrl+shift+down` to select a range of edits

Select edits none | all

1. ☐ + v.addfield /addform/add (9ea401c9)
   node = /saved-interactions/add-speaker, fld = @click

D

Programming 2025

Speakers

Name	Email
Betty Jean Jennings	betty@rand.com

E

Programming 2025

Speakers

Name	Email
Betty Jean Jennings	betty@rand.com
Margaret Hamilton	hamilton@mit.com
Christiane Floyd	floyd@tu-berlin.de

Add speaker

Christiane Floyd, floyd@tu- | Add speaker!

F

Programming 2025

Speakers

- Betty Jean Jennings, betty@rand.com

Budget

- Speakers:/$builtins/count(arg=/speakers)
- Cost:1500
- Total: /$builtins/mul(left=/budget/cost/value/result, right=/budget/count/value/result)

G

Programming 2025

Speakers

Name	Email
Betty Jean Jennings	betty@rand.com
Margaret Hamilton	hamilton@mit.com
Christiane Floyd	floyd@tu-berlin.de

Budget

- Speakers:/$builtins/count(arg=/speakers/body)
- Cost:1500
- Total: /$builtins/mul(left=1500, right=/budget/count/value/result)

Add speaker

e Floyd, floyd@tu-berlin.de | Add speaker!

Figure 1: Organizing a conference using Denicek. The Walkthrough shows construction of a user interface for adding speakers (A, B, C); refactoring of the list and merging edits (D, E); and formulas with schema and code co-evolution (F, G).

3 Walkthrough

In this section, we use Webnicek, our simple web-based prototype, to illustrate the programming experiences enabled by the Denicek substrate (§??). Webnicek is based on a structure editor that supports navigation in the document and issuing of edit commands. We show two formative examples (§??) related to conference organization.

A *Adding a Speaker.* The user starts with an empty document, which is represented as a record. They add a field for each heading and a list `<ul>`, represented by a field named `speakers`. They use the command toolbox to add the first speaker.

B *Creating a User Interface.* To simplify adding of further speakers, the user creates a textbox and a button. They enter new speaker details, add a new `<li>` element to the list and copy the speaker details from the textbox into the new element using source view.

C *Saving the Interaction.* After adding the speaker from the textbox, the user opens history view, selects edits that added the speaker and saves those as the `add-speaker` interaction. They attach this as a handler for the `click` event of the button.

Programming by Demonstration. Denicek implements programming by demonstration (§??) by letting the user to save and replay past interactions, either directly or by attaching them to a UI event.

D *Refactoring Document Structure.* Another user starts with the initial version of the document and turns the list into a table. This is done by invoking a series of commands that change tags, wrap elements, copy and transform values.

E *Merging Edits.* The two document versions can be automatically merged. The refactoring is applied to all existing speakers. New speakers added using the "Add speaker!" button are also automatically added in the new format.

Local-First Collaboration. Denicek's merging reapplies edits from another branch on top of the current history (§??). Merging is asymmetic, but the order does not matter in the above scenario. The same merging operation is used when handling user interaction (§??).

F *Adding Budget Calculation.* A third user adds budget calculation to the initial document. Formulas are represented as special `x-formula` nodes whose arguments are other nodes or references to other nodes or formulas (highlighted on mouse hover).

Incremental Recomputation. Evaluating a formula yields additional edits that augment the document with the result. Those edits are kept at the top of the document history and are removed in case of a conflict (§??), providing incremental recomputation.

G *Merging Formulas.* When the budget calculation is merged with the other edits, references in formulas are automatically updated to point to the new list. Adding new speaker via the "Add speaker!" button invalidates the evaluated result.

Schema Change Control. The substrate understands references in the document and updates them when applying structural edits (§??). Evaluation can replace formulas with values, but also augment them to enable end-user debugging via provenance analysis (§??).

Selector	Notation	
Parent	`..`	Refers to a parent of a node
Field	`field`	Refers to record field of a given name
Index	`#index`	Refers to list element at a given index
Any	`*`	Refers to all children of a list node

Kind *arguments*	
List *tag*, $index_1$, $child_1$, . . ., $index_n$, $child_n$	Ordered list of nodes, addressable by *index*. Renders as `<tag>` with children.
Record *tag*, $field_1$, $child_1$, . . ., $field_n$, $child_n$	Record with children addressable by *field*. Renders as `<tag>` with children.
Reference *selectors*	Reference to another document location. Displays the `selectors` as a link.
Primitivie *string* or *number*	Numerical or textual primitive value. Renders as an HTML text node.

Figure 2: Structure of selectors and document nodes

4 The Denicek Substrate

Denicek represents programs as sequences of edits that construct and transform a computational document. In this section, we describe the structure of documents and edits, as well as the operations that form the backbone of the system and are used to implement the end-user programming experiences discussed in §??.

4.1 Selectors, Documents and Edits

A computational document is a tree, consisting of four kinds of nodes (Fig. 2). References to document locations can be relative or absolute. Both kinds can appear in a document (reference nodes) and absolute references are also used in edits (to denote a target node).

A reference is represented as a sequence of selectors (Fig. 2). The document model assumes that lists are homogeneous and records heterogeneous, and so the Any selector makes it possible to refer to all children of a list, but there is no way to refer to all children of a record. Note that Denicek does not use implicit numerical indices for lists. The index of a new item has to be supplied explicitly. The design choice simplifies merging and is discussed in §??.

Document Edits. The supported document edits and their behavior are listed in Fig. 3. All edits require *target* to which they are applied. Target is an absolute reference not containing the Parent selector. It can contain the Any selector, in which case the edit is applied to multiple nodes simultaneously. Most edits can only be applied to target node(s) of a particular kind.

The edits are designed to allow any transformation of a document through a series of steps whose effect can be tracked by the substrate. As illustrated earlier, Denicek updates references when document structure changes. Fig. 3 distinguishes between edits that keep existing references in a document unchanged (above) and edits that affect references (below).

Renaming a field or wrapping a node require corresponding change to references in the document (§??). When deleting a field to which there is a reference in the document, Denicek rejects the edit. A copy edit can also be rejected, because it is ambiguous whether references referring to the original location should continue referring to the source or should be changed to point to the target of the copying after the edit (a reference cannot refer to two structurally independent locations).

Edit *arguments*		**Target**
Add *target, field, after, node*	Add *node* as a *field* to the specified record *after* a given field.	Record
Append *target, index, after, node*	Append *node* to the end of the specified list *after* a given field.	List
Reorder *target, permutation*	Reorder items of a specified list according to a *permutation*.	List
DeleteItem *target, index*	Delete the item at a given *index* of a specified list.	List
UpdateTag *target, tag*	Change the tag of a specified list or record from to a new *tag*.	Tagged
PrimitiveEdit *target, transform*	Apply primitive *transform* to the specified primitive.	Primitive
RenameField *target, old field, new field*	Rename the field of a specified record from *old* to *new*.	Record
DeleteField *target, field*	Delete the field *field* of a specified record.	Record
WrapRecord *target, tag, field*	Wrap the specified node as a *field* of a new record with *tag*.	Any
WrapList *target, tag, index*	Wrap the specified node as a sole element of a new list with *tag*.	Any
Copy *target, selectors*	Copy nodes(s) from *selectors*, replacing the specified target(s).	Any

Figure 3: Summary of document edit types in Denicek

Automatic Reference Update. There are two situations in which automatic update of references is undesirable. If an edit is applied to a singular element of a list (reference contains the Index selector), references that refer into the list should be unchanged. Although such edits violate the invariant that collections are homogeneous, they typically do so only temporarily, for example during construction of a new list item (checking such edits is discussed in §??).

Documents can also contain values that can be of multiple different kinds (i.e., a union type). In such case, references should not be updated when the kind of the value changes. For example, a formula may be either unevaluated or evaluated. As discussed in §??, evaluation involves wrapping the formula node, but this should not affect references to the formula. To support those cases, edits that normally affect selectors (Fig. 3, below) have a *reference behavior* field that can be set to disable automatic reference updating. This annotation is required when the target contains the *Index* selector.

No Conditional Edits. An important aspect of the design is that the effect an edit has on references inside the document does not depend on the current value of the document. This makes it possible to define merging solely in terms of edits, without reference to current document state. (The effect Copy has on the document structure depends on the existing document structure, but not on its value).

This design choice makes it impossible to encode computational logic directly in the edits (e.g., through conditional edits). As we will see in §??, such logic can be provided as an additional mechanism on top of the underlying Denicek substrate.

4.2 Primitive Operations

Denicek provides three primitive operations. A sequence of edits can be applied to a document, two sequences of edits can be merged and also checked for conflicts. Denicek identifies edit histories by a (git-like) hash, computed from the hash of the parent and the current edit. The hash is used to identify a common shared part of the history during conflict checking and merging.

Applying Edits. When applying an edit, Denicek locates the target node and transform it according to the edit. If the edit can affect references in the document (Fig 3, below), Denicek updates matching references in the document according to the rules shown in Fig. **??** provided that the reference behavior of the edit is not set to disable the updating (this is required when the target reference contains the Index selector). Reference update behavior for WrapRecord and WrapList differs in that the updated references as a result of WrapList use the All selector. Although WrapList specifies the index to be used for the newly created list item, we assume that the operation introduces a homogeneous list and the new reference should point to all eventual list items.

Also note that affected references in the document may be more specific than the edit target. For example, if we rename `old` to `new` at `/foo/*`, a reference `/foo/3/old` will become `/foo/3/new`. (A reference in the document cannot be more general, because such edit would have to contain the Index selector and this would require setting reference behavior to not trigger reference update.) Finally, if the document contains a reference that would be invalidated by the Copy or DeleteField edit, the edit is rejected.

Merging Edit Histories. Merging edit histories is used when two users edit document independently, but also when replaying edits in programming by demonstration. The operation works on two edit histories, E, E_1 and E, E_2, that have a shared prefix E. The merging operation is akin to git rebase. It turns edits E_2 into edits E_2' that can be reapplied on top of the other edit history, resulting in a new history E, E_1, E_2'. Note that the operation is not symmetric. If there are conflicts among the edits in E_1 and E_2, the result of E, E_1, E_2' will differ from the result of E, E_2, E_1'. We return to this problem in the next section when discussing conflict detection.

The key operation that enables such reconcilliation takes two individual edits that occurred indpendently, e_1 and e_2, and produces a sequence of edits $e_2', e_2'', \ldots$ that can be applied after e_1 and have the effect of e_2, modified to respect the effects of e_1. There are two aspects of such reconcilliation:

(1) *Apply to Newly Added.* If e_2 is adding new nodes to the document, but e_1 modified the document through a selector that would also affect the new nodes added by e_2, we need to apply the transformation represented by e_1 to the nodes newly added by e_2. This is done by generating an additional edit, to be applied after e_2, that is based on e_1 but targets only the newly added nodes (more details can be found in §**??**).

(2) *Transform Matching References.* If e_2 targets a node that is inside a node whose structure is changed by e_1, the target reference in e_2 is updated in a way that corresponds to the new structure. This is done using the rules shown in Fig. **??**, that apply when transforming references inside a document, although we also support the case when $e1$ is Copy (see §**??** for details).

RenameField *target, old field, new field* – Replace Field for matching references.
`/target/old_field/nested` $\Rightarrow$ `/target/new_field/nested`

WrapRecord *target, tag, field* – Insert extra Field selector after matching prefix.
`/target/nested` $\Rightarrow$ `/target/field/nested`

WrapList *target, index, tag* – Insert extra All selector after matching prefix.
`/target/nested` $\Rightarrow$ `/target/*/nested`

Figure 4: How document edits transform references

no longer

Applying an edit to a newly added node involves a number of cases. When the new node is added using Add or Copy, but the target location is modified, this is an unresolvable conflict (because those two operations would overwrite existing nodes). When the new node is added as a new list item using Append, it can be transformed (by applying the other edit direclty to the new node).

Finally, when the new node is added using AppendFrom, it is copied from another document location. We cannot transform the node in the soource location (this would have unintended effects) or after adding it (we do not know its new index because of aforementioned *non-conditionality* of edits). Denicek reconcilliates such edits by first copying the source node to a new temporary location, transforming it there and then using AppendFrom from the temporary location.

todo

Removing all conflicting edits is sometimes useful, but there are many cases where conflicting edits, in the above sense, can be reconcilliated.

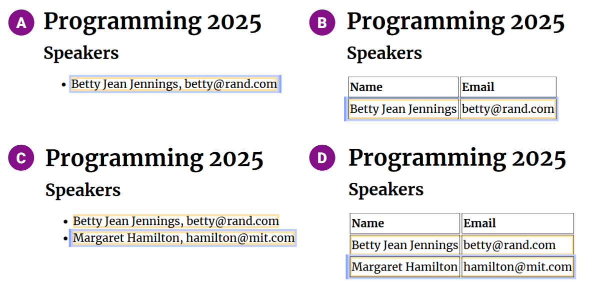
If the edits in E_1 and E_2 conflict, the results of applying E, E_1, E_2 and E, E_2, E_1 would be different. We can, however, construct edit histories E_1' and E_2' such that the results of applying E, E_1, E_2' and E, E_2, E_1' are the same.

Conflict Detection. Two edits are conflicting if the order in which they are applied matters. This is the case if they target the same node, or if one targets a node that is nested inside the node targetted by the other. For edits with dependencies (AppendFrom and Copy), a conflict also occurs if the other edit modifies the dependency.

Given two edit histories, one way to resolve conflicts is by removing (or marking as disabled) all edits from one of the histories that conflict with edits done by the other history. To do this, we first collect all targets of edits in the other history. We then iterate over edits from the first history to see if they depend on or target any of the affected targets. If an edit is removed, its target reference is added to the set of affected targets, so that other subsequent edits that depend on its original result are also removed.

5 Programming Experiences Implementation

Denicek follows the *naive realism* [?] principle and makes the entire document visible to the user, although parts can be collapsible or hidden using CSS.



A: Programming 2025

Speakers

- Betty Jean Jennings, betty@rand.com

B: Programming 2025

Speakers

Name	Email
Betty Jean Jennings	betty@rand.com

C: Programming 2025

Speakers

- Betty Jean Jennings, betty@rand.com
- Margaret Hamilton, hamilton@mit.com

D: Programming 2025

Speakers

Name	Email
Betty Jean Jennings	betty@rand.com
Margaret Hamilton	hamilton@mit.com

Figure 5: Merging of two independently done sequences of edits. Two ways of merging B and C result in the same D.

[webnicek?]

The key claim of this paper is that the computational substrate described in the previous section makes it easy to support a range of experiences that make programming more concrete, collaborative and interactive. In this section, we describe how to use the substrate to support local-first collaborative editing (§??), programming by demonstration (§?? and §??), schema and code co-evolution (§??), incremental recomputation (§?? and §??), provenance tracking (§??) and concrete programming via managed copy & paste (§??). We describe the programming experience in isolation in this section. Next section provides a more comprehensive evaluation through a case study that combines multiple of them together.

5.1 Local-First Collaborative Editing

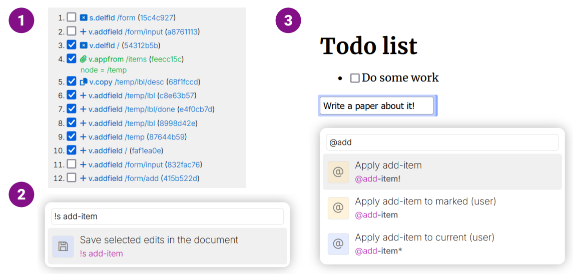
The Denicek representation enables *local-first* collaboration [?], as illustrated in §3E. If a document is edited by multiple users, they can each make edits to their local copy and eventually merge document variants using the operation to merge edit histories.

Merging of histories behaves akin to git rebase in that it keeps a linear history. Synchronization in a distributed system thus requires first reapplying local edits on top of the remote history, before updating the remote history. Denicek thus implements the *convergence model* of document variants [?], i.e., the user cannot, for example, maintain their own local document structure and import new data from another variant (this would require an inverse of the edit reconcilliation operation).

Merging of edit histories is not symmetric. Document D in Fig. ?? can be obtained either by appending C' (produced by the edit reconcilliation operation) on top of A, B or by appending B' on top of A, C. Although the two edits are conflicting, C only affects data and B only affects structure and so the resulting document is the same in both cases. However, the histories differ. In the first case, the new node added by the Append edit is transformed (from primitive string to a record representing table row). In the second case, the structural transformations are automatically applied to all rows. As discussed in §??, conflicts during merging can be resolved either by removing conflicting edits or by letting the later edits overwrite the former ones.

5.2 Programming By Demonstration

In *programming by demonstration* [?], the user demonstrates a task to the system and the system then repeats it, directly or in a generalized way. To use direct repetition with Denicek (Fig. ??), the



```
1.  ☐ s.delfld /form (15c4c927)
2.  ☐ + v.addfield /form/input (a8761113)
3.  ☑ v.delfld / (54312b5b)
4.  ☑ v.appfrom /items (feecc15c)
      node = /temp
5.  ☑ v.copy /temp/lbl/desc (68f1fccd)
6.  ☑ + v.addfield /temp/lbl (c8e63b57)
7.  ☑ + v.addfield /temp/lbl/done (e4f0cb7d)
8.  ☑ + v.addfield /temp/lbl (8998d42e)
9.  ☑ + v.addfield /temp (87644b59)
10. ☑ + v.addfield / (faf1ea0e)
11. ☐ + v.addfield /form/input (832fac76)
12. ☐ + v.addfield /form/add (415b522d)
```

!s add-item

Save selected edits in the document
!s add-item

Todo list

- ☐ Do some work

Write a paper about it!

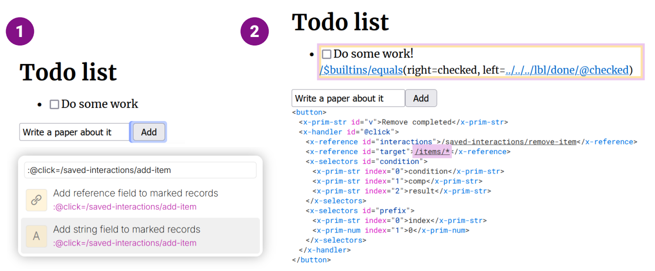
@add

Apply add-item
@add-item!

Apply add-item to marked (user)
@add-item

Apply add-item to current (user)
@add-item*

Figure 6: Programming by demonstration is implemented by selecting edits from the document history (1), saving them in the document (2) and replaying them (3).



Todo list

- ☐ Do some work

Write a paper about it

:@click=/saved-interactions/add-item

Add reference field to marked records
:@click=/saved-interactions/add-item

Add string field to marked records
:@click=/saved-interactions/add-item

Todo list

- ☐ Do some work!
/$builtins/equals(right=checked, left=../../../lbl/done/@checked)

Write a paper about it

```
<button>
  <x-prim-str id="v">Remove completed</x-prim-str>
  <x-handler id="@click">
    <x-reference id="interactions">/saved-interactions/remove-item</x-reference>
    <x-reference id="target">/items/*</x-reference>
    <x-selectors id="condition">
      <x-prim-str index="0">condition</x-prim-str>
      <x-prim-str index="1">comp</x-prim-str>
      <x-prim-str index="2">result</x-prim-str>
    </x-selectors>
    <x-selectors id="prefix">
      <x-prim-str index="0">index</x-prim-str>
      <x-prim-num index="1">0</x-prim-num>
    </x-selectors>
  </x-handler>
</button>
```

Figure 7: Using programming by demonstration to define a UI. The "Add" button (1) replays edits; the "Remove completed" button (2) modifies target and specifies a condition.

user can select edits from the edit history, name them and replay them. In case of general-purpose document editing (in our formative prototype), this requires certain forethought (e.g., constructing a new list item in a temporary document field before appending it), but as illustrated in §??, the mechanism is effective in a restricted domain such as data wrangling [?].

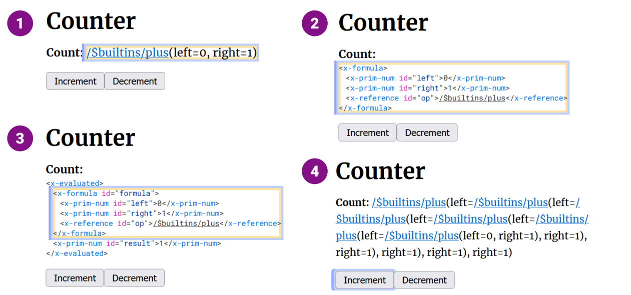
There are two notable aspects of our implementation. First, Denicek saves edits in the document itself (by representing individual edits as nodes and storing them in list inside a `<saved-interactions>` field). This means that no other implementation mechanism outside of the system is needed and also that the stored edits can be modified by the user (or tools working with the document).

Second, to replay edits, Denicek does not simply append them on top of the current history. When saving edits, it records the hash of the history at the time of saving. When replaying edits, it appends the saved edits to the top of the original history (at the time of saving) and merges this new sequence with the current history. This pushes the saved edits through all subsequent edits made by the user. The result can be seen in Fig. 1 (E), where a newly added speaker is transformed from a primitive string to a table row.

The implementation also has to account for the case where edits saved in the document are themselves transformed (when they are reconcilliated with other edits during merging). In this case, Denicek updates the saved edits (as discussed in §??, this would not be needed if the history was stored as a graph, akin to ordinary git merging rather than rebasing).

5.3 Interactive User Interfaces

Programming by demonstration can be used to define interactive elements in the document. In the simple case (Fig. ?? (1)), the `click`



1 Counter

Count: /$builtins/plus(left=0, right=1)

2 Counter

Count:

```
<x-formula>
  <x-prim-num id="left">0</x-prim-num>
  <x-prim-num id="right">1</x-prim-num>
  <x-reference id="op">/$builtins/plus</x-reference>
</x-formula>
```

3 Counter

Count:

```
<x-evaluated>
 <x-formula id="formula">
  <x-prim-num id="left">0</x-prim-num>
  <x-prim-num id="right">1</x-prim-num>
  <x-reference id="op">/$builtins/plus</x-reference>
 </x-formula>
 <x-prim-num id="result">1</x-prim-num>
</x-evaluated>
```

4 Counter

Count: /$builtins/plus(left=/$builtins/plus(left=/$builtins/plus(left=/$builtins/plus(left=/$builtins/plus(left=/$builtins/plus(left=0, right=1), right=1), right=1), right=1), right=1), right=1)

Figure 8: Increment wraps the existing count in a formula that adds 1 to the previous value (1), (2). Evaluation produces the count (3), which is invalidated on subsequent clicks (4).

event handler is set to a reference to a sequence of edits saved in the document. Clicking the button executes the edits using the mechanism discussed in §??, i.e., by appending them to a history at the time of saving and merging them with the current history.

The Denicek substrate can be used to generalize the interactions saved through programming by demonstration. Our prototype illustrates this option in a limited way. As shown in Fig. ?? (2), a button to remove all completed TODO items can be generalized from the `remove-item` interaction, which removes the list item at the index 0. In addition to the saved interaction, we manually specify (in the source view) that the edits should be applied to all elements selected by the `/items/*` selector, instead of the original `/items/0` selector (prefix) and that the edits should only be applied to elements for which the formula (which tests if the checkbox is checked) specified by a relative selector `./condition/comp/result` evaluates to true.

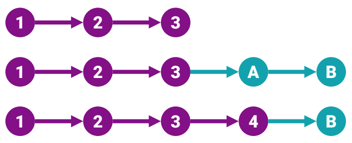
Generalization Heuristic. Specifying generalization manually is cumbersome. Programming by demonstration systems typically implement heuristic for generalization [?] infers and suggests such generalizations. When integrated into a system based on Denicek, such heuristic may also be able to automatically add a formula based on positive and negative examples [?] (selected and deselected items).

5.4 Formula Language and Evaluation

Denicek documents can contain formulas inspired by the spreadsheet paradigm [?]. Formulas can specify richer computations than what can be expressed using document edits. Formulas do not transform the document and their results are transient, although their evaluation also leverages the substrate operations, namely merging of edit histories.

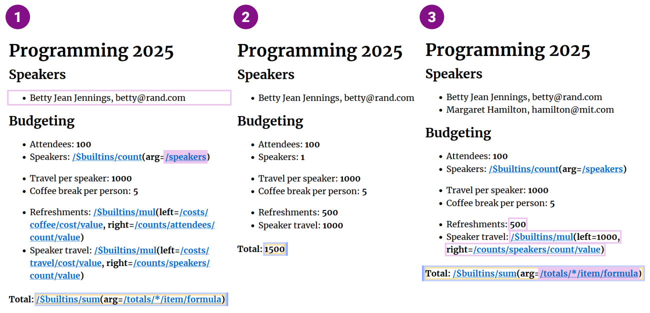
As illustrated in Fig. ??, formulas are represented as document nodes with a special tag (`<x-formula>`). They are recognized by a formula evaluator and rendered in a special way (❶ and ❹), but they are created using ordinary edits and the Denicek substrate treates them as stnadard nodes.

To evalaute formulas, the formula evaluator generates edits that turn the `<x-formula>` record into `<x-evaluated>` (❷ and ❸), which keeps the previous formula state in the `formula` field and the evaluation result in the `result` field. (Keeping the previous state of the formula is not necessary, but it enables provenance analysis as discussed in §??.) The way edits generated by evaluation are merged with the document is discussed in §??



$$\begin{tikzcd}
1 \arrow[r] & 2 \arrow[r] & 3 & & \\
1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & A \arrow[r] & B \\
1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 4 \arrow[r] & B
\end{tikzcd}$$

Figure 9: Evaluated edits (A, B) are kept at the top of the history. When they conflict with ordinary edits (4), evaluated edits are removed (A).



1

Programming 2025

Speakers

- Betty Jean Jennings, betty@rand.com

Budgeting

- Attendees: 100
- Speakers: /$builtins/count(arg=/speakers)
- Travel per speaker: 1000
- Coffee break per person: 5
- Refreshments: /$builtins/mul(left=/costs/coffee/cost/value, right=/counts/attendees/count/value)
- Speaker travel: /$builtins/mul(left=/costs/travel/cost/value, right=/counts/speakers/count/value)

Total: /$builtins/sum(arg=/totals/*/item/formula)

2

Programming 2025

Speakers

- Betty Jean Jennings, betty@rand.com

Budgeting

- Attendees: 100
- Speakers: 1
- Travel per speaker: 1000
- Coffee break per person: 5
- Refreshments: 500
- Speaker travel: 1000

Total: 1500

3

Programming 2025

Speakers

- Betty Jean Jennings, betty@rand.com
- Margaret Hamilton, hamilton@mit.com

Budgeting

- Attendees: 100
- Speakers: /$builtins/count(arg=/speakers)
- Travel per speaker: 1000
- Coffee break per person: 5
- Refreshments: 500
- Speaker travel: /$builtins/mul(left=1000, right=/counts/speakers/count/value)

Total: /$builtins/sum(arg=/totals/*/item/formula)

Figure 10: Budget calculation based on the number of speakers (1) and the result (2). When speaker is added (3), only the results of affected formulas are invalidated.

The counter example shown in Fig. ?? illustrates the interaction between formulas and programming by demonstration. To implement a counter, the Increment and Decrement buttons wrap the current counter value in a formula that adds or subtracts 1.

5.5 Incremental Recomputation

As illustrated in Fig. ??, Denicek supports incremental re-computation. In the example, the cost of speaker travel depends on the number of speakers, but the cost of refreshments depends only on two constants. Edit that adds a speaker only invalidates the former.

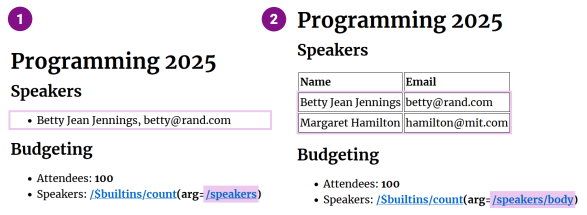
The evaluation mechanism is illustrated in Fig. ??. When the formulas are evaluated, Denicek generates *evaluated edits* that are appended to the top of the history. When subsequent edits are made, they are appended after all non-evaluated edits and the evaluated edits are pushed through the newly added edits. If the edits conflict (according to Denicek's conflict detection), affected evaluated edits and edits that depend on them are dropped.

Live Programming. In the exploratory prototype, Denicek does not automatically evaluate formulas. This makes it easier to see and understand the evaluation mechanism, but a realistic system based on the substrate would likely automatically evaluate formulas to provide a live programming experience [? ?].

5.6 Schema Code Co-evolution

The problem of schema evolution is that, when the schema used in a database or a program evolves, data and code that depend on the schema need to evolve correspondingly. The problem is well-known in database systems [?] and has recently been studied in the context of programming systems [?].

Although Denicek does not explicitly track document structure (schema or type), all documents have an implicit structure and some


1

Programming 2025

Speakers

- Betty Jean Jennings, betty@rand.com

Budgeting

- Attendees: **100**
- Speakers: /$builtins/count(arg= /speakers)

2 Programming 2025

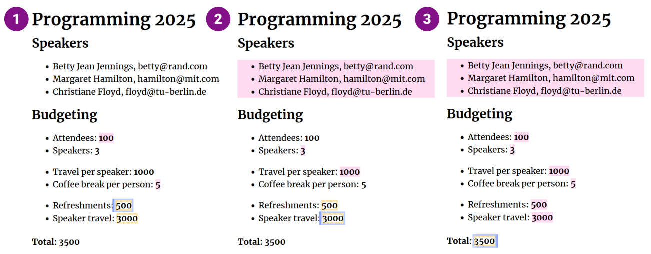
Speakers

Name	Email
Betty Jean Jennings	betty@rand.com
Margaret Hamilton	hamilton@mit.com

Budgeting

- Attendees: **100**
- Speakers: /$builtins/count(arg= /speakers/body)

Figure 11: When an edit changes the document structure, references in formulas are updated accordingly.


1 Programming 2025

Speakers

- Betty Jean Jennings, betty@rand.com
- Margaret Hamilton, hamilton@mit.com
- Christiane Floyd, floyd@tu-berlin.de

Budgeting

- Attendees: **100**
- Speakers: **3**
- Travel per speaker: **1000**
- Coffee break per person: **5**
- Refreshments: **500**
- Speaker travel: **3000**

Total: 3500

2 Programming 2025

Speakers

- Betty Jean Jennings, betty@rand.com
- Margaret Hamilton, hamilton@mit.com
- Christiane Floyd, floyd@tu-berlin.de

Budgeting

- Attendees: **100**
- Speakers: **3**
- Travel per speaker: **1000**
- Coffee break per person: **5**
- Refreshments: **500**
- Speaker travel: **3000**

Total: 3500

3 Programming 2025

Speakers

- Betty Jean Jennings, betty@rand.com
- Margaret Hamilton, hamilton@mit.com
- Christiane Floyd, floyd@tu-berlin.de

Budgeting

- Attendees: **100**
- Speakers: **3**
- Travel per speaker: **1000**
- Coffee break per person: **5**
- Refreshments: **500**
- Speaker travel: **3000**

Total: 3500

Figure 12: Using provenance tracking to highlight document parts that contributed to the calculation of refreshments costs (1), travel costs (2) and all costs (3).

of the edits transform the document structure. As discussed in §??, Denicek automatically updates reference nodes in the document in this case. This enables a form of schema code co-evolution [?]. The formulas embedded in Denicek documents use reference nodes to refer to both data sources (in the document) and the results of other computations. Consequently, if the document structure changes, the formulas are automatically updated.

Consider the example in Fig. ??. The original list `<ul>` is turned into `<tbody>` using UpdateTag and wrapped inside `<table>` with a field `body` using WrapRecord. For the latter, Denicek updates the reference accordingly turning the original `/speaker` reference in the formula into `/speaker/body`.

As noted earier, evaluated edits transform the formula structure (wrapping it in the `<x-evaluated>` record). However, they are marked as non-structural and so references to formulas are not transformed when the formula is evaluated (otherwise, references to evaluated formulas would not point to the evaluated result, but to the original unevaluated definition).

5.7 Provenance tracking

5.8 Managed copy & paste

6 Design Discussion

LISTS vs RECORDS - are they really different? also MRDT idea - needed for both

PRINCIPLES * merging of edits should not require looking at current document (non-computationality) * two ways of pushing edits should be equivalent, i.e.

push e1 e2 e3 over b1 b2 b3 can be done via push e1 over b1 = e1'; over b2 = e1"; over b3 = e1"' then push e2, then push e3 or push e1, e2, e3 over b1, then over b2, then over b3 (implies that no "extra" edits can be appended later)

USE MRDTs for list order!

DIFFICULTIRES * if you are not able to refer to newly added items, you cannot reasonably do PBD => need non-numerical indices

how to transform appends? * apply to added? (does not work for copy) * copy to doc and edit? (good because edits are isolated from other things, but messy code!) (if we edit in doc, it works and we also do not need indices for ListAppend) * keep nested edits for append op? also does not work, because of the tricky case

Tricky case is: - add new list item - then set its fields (this breaks things because if we push "add" through structural edits, they will not affect the new values added in the second step)

TODO: Delete ListAppend

// cf shadow dom v reactu // something like google forms!

TAG selectors - sound nice, but they break (you cannot tell if two selectors overlap e.g. when there is index >< tag)

save graph of edits rather than list (would make replaying of saved edits easier because the histhash never changes)

Although Denicek does not explicitly track document structure (or schema, or type), all documents have an implicit structure.

c.f. notes

memory mapped graphics etc.

unifying lists and records?

AppendFrom - hard to avoid! because recorded edits cannot know length (but having -1 in index would break too)

7 Case study

(data science environment)

8 Evaluation

how well did Denicek work for implementing Datnicek

9 Discussion

9.1 Heuristic evaluation

9.2 Limitations

9.3 Future work

use type system to check temporarily invalid state

ohter formal things * non-conditionality of edits * show that they can transform document from any to any without removing/readding values

PROPERTIES - can change any document to any other - sure, via remove add - but also more semantically (if it contained all values, can we do it without removing and adding them?)

References

[1] Marc T.P. Adam, Shirley Gregor, Alan Hevner, and Stefan Morana. Design science research modes in human-computer interaction projects. *AIS Transactions on Human-Computer Interaction*, 13(1):1–11, 2021.

[2] Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. Structured editing for all: Deriving usable structured editors from grammars. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23. Association for Computing Machinery, 2023.

[3] Weihao Chen, Xiaoyu Liu, Jiacheng Zhang, Ian Iong Lam, Zhicheng Huang, Rui Dong, Xinyu Wang, and Tianyi Zhang. MIWA: mixed-initiative web automation for better user control and confidence. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software Technology, UIST 2023, San Francisco, CA, USA*, pages 75:1–75:15. ACM, 2023.

[4] James Cheney, Stephen Chong, Nate Foster, Margo Seltzer, and Stijn Vansummeren. Provenance: a future history. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 957–964, New York, NY, USA, 2009. Association for Computing Machinery.

[5] Allen Cypher and Daniel Conrad Halbert. *Watch what I do: programming by demonstration.* MIT press, 1993.

[6] Andrea A. diSessa and Harold Abelson. Boxer: A reconstructible computational medium. *Commun. ACM*, 29(9):859–868, 1986.

[7] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–12. Association for Computing Machinery, 2020.

[8] Jonathan Edwards. First class copy & paste. Technical Report MIT-CSAIL-TR-2006-037, Massachusetts Institute of Technology, 2006.

[9] Jonathan Edwards and Tomas Petricek. Interaction vs. abstraction: Managed copy and paste. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments*, pages 11–19, 2022.

[10] Jonathan Edwards, Tomas Petricek, Tijs van der Storm, and Geoffrey Litt. Schema evolution in interactive programming systems. *The Art, Science, and Engineering of Programming*, 9(?):1–34, 2025.

[11] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, August 2012.

[12] Joshua Horowitz and Jeffrey Heer. Engraft: An api for live, rich, and composable programming. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, UIST '23, New York, NY, USA, 2023. Association for Computing Machinery.

[13] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. Technical dimensions of programming systems. *The Art, Science, and Engineering of Programming*, 7(13):1–59, 2023.

[14] Joel Jakubovic and Tomas Petricek. Ascending the ladder to self-sustainability: Achieving open evolution in an interactive graphical system. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 240–258, 2022.

[15] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, page 3363–3372, New York, NY, USA, 2011. Association for Computing Machinery.

[16] Alan C. Kay. The early history of smalltalk. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, page 69–95. Association for Computing Machinery, 1993.

[17] Stephen Kell. Unix, plan 9 and the lurking smalltalk. In *Reflections on Programming Systems: Historical and Philosophical Aspects*, pages 189–213. Springer, 2018.

[18] Stephen Kell and J. Ryan Stinnett. Source-level debugging of compiler-optimised code: Ill-posed, but not impossible. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '24, page 38–53. Association for Computing Machinery, 2024.

[19] Eugen Kiss. Comparison of object-oriented and functional programming for gui development. Master's thesis, Leibniz Universität Hannover, 2014.

[20] Cory Kissinger, Margaret Burnett, Simone Stumpf, Neeraja Subrahmaniyan, Laura Beckwith, Sherry Yang, and Mary Beth Rosson. Supporting end-user debugging: what do users want to know? In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '06, page 135–142. Association for Computing Machinery, 2006.

[21] Martin Kleppmann, Adam Wiggins, Peter Van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 154–178, 2019.

[22] Clemens Nylandsted Klokmose, James R Eagan, and Peter van Hardenberg. Mywebstrates: Webstrates as local-first software. In *UIST'24: Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. ACM, 2024.

[23] Amy J Ko and Brad A Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158, 2004.

[24] Amy J. Ko and Brad A. Myers. Finding causes of program output with the java whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, page 1569–1578. Association for Computing Machinery, 2009.

[25] Eva Krebs, Patrick Rein, Joana Bergsiek, Lina Urban, and Robert Hirschfeld. Probe log: Visualizing the control flow of babylonian programming. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming*, pages 61–67, 2023.

[26] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 542–553, New York, NY, USA, 2014. Association for Computing Machinery.

[27] Germán Leiva, Jens Emil Grønbæk, Clemens Nylandsted Klokmose, Cuong Nguyen, Rubaiat Habib Kazi, and Paul Asente. Rapido: Prototyping interactive ar experiences through programming by demonstration. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 626–637, 2021.

[28] Sorin Lerner. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–7. Association for Computing Machinery, 2020.

[29] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. Peritext: A crdt for collaborative rich text editing. *Proc. ACM Hum.-Comput. Interact.*, 6(CSCW2), November 2022.

[30] Geoffrey Litt, Peter van Hardenberg, and Henry Orion. Project cambria: Translate your data with lenses. https://www.inkandswitch.com/cambria.html, 2020. Accessed: 2020-10-01.

[31] Sean McDirmid. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 53–62. Association for Computing Machinery, 2013.

[32] Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, page 161–174, USA, 2001. USENIX Association.

[33] Brad A. Myers, Amy J. Ko, and Margaret M. Burnett. Invited research overview: end-user programming. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '06, page 75–80. Association for Computing Machinery, 2006.

[34] Brad A. Myers, Richard McDaniel, and David Wolber. Programming by example: intelligence in demonstrational interfaces. *Commun. ACM*, 43(3):82–89, March 2000.

[35] Bonnie A. Nardi and James R. Miller. The spreadsheet interface: A basis for end user programming. In *Proceedings of the IFIP TC13 Third Interational Conference on Human-Computer Interaction*, INTERACT '90, page 977–983, NLD, 1990. North-Holland Publishing Co.

[36] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. Filling typed holes with live guis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 511–525, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 86–99, New York, NY, USA, 2017. Association for Computing Machinery.

[38] Addy Osmani, Sindre Sorhus, Pascal Hartig, and Stephen Sawchuk. Todomvc: Helping you select an MV* framework. https://todomvc.com/, 2024. Accessed: 2024-12-12.

[39] Roly Perera, Umut A Acar, James Cheney, and Paul Blain Levy. Functional programs that explain their work. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 365–376, 2012.

[40] Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. Linked visualisations via galois dependencies. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022.

[41] Tomas Petricek. Foundations of a live data exploration environment. *The Art, Science, and Engineering of Programming*, 4(8):1–37, 2020.

[42] Tomas Petricek. Foundations of a live data exploration environment. *The Art, Science, and Engineering of Programming*, 4(8):1–37, 2020.

[43] Erhard Rahm and Philip A. Bernstein. An online bibliography on schema evolution. *SIGMOD Rec.*, 35(4):30–31, December 2006.

[44] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. Babylonian-style programming: Design and implementation of a general-purpose editor integrating live examples into source code. *The Art, Science, and Engineering of Programming*, 3(9):1–39, 2019.

[45] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. Exploratory and live, programming and coding: A literature study comparing perspectives on liveness. *The Art, Science, and Engineering of Programming*, 3(1):1–33, 2019.

[46] Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. Imperative functional programs that explain their work. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–28, 2017.

[47] Guy L. Steele and Richard P. Gabriel. The evolution of lisp. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, page 231–270. Association for Computing Machinery, 1993.

A Formative Examples

The design the Denicek substrate, we identified five formative examples shown in Fig. **??**. The examples range from established industry benchmarks (Todo and Counter apps) to problems posed as schema change challenges [?]. The Denicek substrate then co-evolved with Webnicek, a simple web-based programming environment built (as directly as possible) on top of the substrate and was used to solve implement the formative examples.

Many of the formative examples include a small programming challenge, such as adding user interface to add a new speaker, a new list item or modify the count. Our aim was for the substrate to enable solving those through programming by demonstration. Programming by Demonstration is often used in data wrangling [? ? ?]. Our Hello World example is only a minimalistic illustration of such use, loosely inspired by earlier work [?].

B Merging Edit Histories

Recall that merging takes two edit histories, E, E_1 and E, E_2, transforms edits E_2 into E_2' that can be reapplied on top of the first history resulting in E, E_1, E_2'. The key operation takes two individual edits, e_1 and e_2 and produces a sequence of edits $e_2', e_2'', \ldots$ that can be applied after e_1, and combine the two edits. This section provides details about the two aspects of this operation.

B.1 Apply to Newly Added

Assume that edits e_1 and e_2 occurred independently. We want to modify e_2 so that it can be placed after e_1. If the edit e_2 added new nodes to the document that the edit e_1 would affect, we generate an additional edit that apply the transformation of e_1 to the newly added nodes (and only to those).

The only edits that add new document nodes are Add, Append, Copy and so we consider this case if the edit e_2 is one of those. If so, we check whether the target of e_1 is within the target of e_2, i.e., the list of selectors that forms the target of e_2 is a prefix of the list of selectors that forms the target of e_1.

Along the way, we compute a *more specific prefix*. If the target of e_1 contains the All selector, it can be matched against a specific Index selector in the target of e_2 (if the selector of e_1 is more specific than that of e_2, the targets are not matched). We then replace the original prefix in e_1 with the *more specific prefix* that contains Index selector in places where the original edit contained All. This way, we obtain e_1' which is a focused version of e_1 that applies only to the nodes newly added by e_2. The edit e_2 thus becomes a pair of edits e_2, e_1'. The final document will contain edits e_1, e_2, e_1' – that is, it will first apply the edit e_1 to nodes already in the document, then add new nodes and then apply the transformation represented by e_1 to the newly added nodes.

B.2 Transform Matching References

As above, assume that edits e_1 and e_2 occurred independently. We want to modify e_2 so that it can be placed after e_1. If e_1 is any of the three edits listed in Fig. **??** (RenameField, WrapRecord, WrapList), we collect all references that appear inside e_2 (the target, the source of Copy and any references occurring in the nodes added by Add or Append). If the target of e_1 is a prefix of any of those references, we update the references accordingly and obtain a new edit e_2'. Note

Counter [?] – Counter with increment and decrement buttons.
The current count is represented by a formula that is modified by the buttons.
The user can inspect the evaluation trace to see how the count was modified.
Programming by Demonstration, Incremental Recomputation, End-User Debugging

Todo [?] – App with controls to add an item and remove all completed.
Adding an item must correctly merge with independently added functionality to compute which items are completed and remove them based on a formula result.
Programming by Demonstration, Local-First Collaboration, Incremental Recomputation

Conference List [?] – Manage a list of invited conference speakers.
Adding speakers to a list through an in-document user interface merges with refactoring that turns the list into a table and separates name from an email.
Local-First Collaboration, Programming by Demonstration

Conference Budget [?] – Calculate budget based on a speaker list.
References are updated when the list is refactored. Only affected formulas are recomputed and the user can view elements on which the result depends.
Local-First Collaboration, Incremental Recomputation, End-User Debugging

Hello World [?] – Normalize the capitalization of two word messages.
An operation to normalize the text in a list item can be recorded and applied to all list items or, alternatively, applied directly to all list items.
Programming by Demonstration

Figure 13: Formative examples used in Denicek design

that it would be an error to match specific Index in e_1 with more general All in e_2, but this cannot happen – reference updating is not done when the target of e_1 contains Index.

Now consider the case when e_1 is Copy and the edit e_2 targets a node that is the source node of the copy operation (or any of its children). In this case, it is reasonable to require that the edit e_2 is applied to both the source and the target of the copy. (This is required by the refactoring done in the Conference Budget example.) We handle the case by creating a copy of e_2 with transformed selectors (target and, if e_2 is also Copy, also its source). To transform the selectors, we replace the prefix formed by the source of the Copy by a new prefix, formed by the target target of the Copy. We then add the new operation as e_2' if at least one of its selectors was transformed (typically target, but possibly also source).

xx

introduction
background
related work
design process / goals
case study
formative research
formative study
analysis
system
implementation
evaluation / heuristic evaluation
discussion and limitations

C Introduction old

The computational substrate using which software is built determines the capabilities that the software can provide. An imperative substrate that views programs as instructions modifying bytes in memory makes it almost impossible to allow end-user inspection or reprogramming of running software.

substrate as defined by [?]

A computational substrate defines what software is built from. This may be objects as in Smalltalk, lists as in Lisp, or memory with data and code as in UNIX/C. The different substrates enable different kinds of programming experiences. For example, object-oriented programming has historically been linked to the development of graphical user interfaces (where objects can correspond to elements on the screen). It has also enabled the development of visual programming environments such as the Alternate Reality Kit, based on message sending between objects.

In principle, any computational substrate can be used to develop any programming experience, but the greater the impedance mismatch between the substrate and the desired experience, the more difficult it will be to provide the experience and combine it with the rest of the system and other programming experiences developed for the system. (One can implement support for programming-by-demonstration using C/C++, for example as part of a game scripting engine, but it will not work with the rest of the ordinary C/C++ ecosystem.)

Medium is the message

C.1 Substrate

The question asked in this paper is, what would be the ideal programming substrate for supporting a range of programming experiences that make programs more collaborative, transparent and allows for a gradual transition from non-programmer to a programmer. We want a programming substrate that makes it easy to develop programming experiences such as:

- *Programming by demonstration* – Allow non-programmers to construct simple programs by performing examples of the expected behaviour. [?].
- *Local-first collaboration* – Multiple users should be able to use and modify a single program, preferrably without requiring a central server. [?]
- *Provenance tracking* – The execution of the program should leave an understandable trace that lets the user understand why program resulted in a particular result.
- *Schema evolution [extra-ish]* – When the user evolves the structure of the program, data and code should co-evolve automatically to match the new structure.
- *Notational freedom [extra-ish?]* – Allow users to adapt the program using a notation that suits them and is appropriate for the programming task at hand. [Joel]
- *Concrete programming [extra?]* – It should be possible to reuse parts of program or program logic without constructing abstractions, for example by managed copy & paste.[? ?]

substrate as defined by [?]

[? ?] [?]

[? ?] [? ? ?] [? ?] [?]

Joel's definition of substrate in Onward! Bret Victor talk https://www.youtube.com/watch?v=ef2jpjTEB5U

In what ways is a substrate "natural"?

thinglab - create line by cloning, it sticks to mouse pointer, clicking sticks it to something else squeak - has all the browsers (method search...)

computational substrate how it differs from computational media? more low-level - media suggests that there it comes

D The whatever system

D.1 Document + Edits

defines

- selectors
- nodes
- edits

D.2 Walkthrough

* todo list? (or counter, but that is a bit boring)

E Themes

* programming by demonstration - binding interactions to gui elements (event handlers) * provenance tracking - Amy Ko's whyline, Probe Log by HPI, enables linked visualizations * merging of edit histories / collaborative editing - bonus - can share restricted link to allow users fill out forms (allow partial edits only / def by selector?) * scehma change - change data & code accordingly * everything is an edit - interaction with the GUI - evaluation? tbd * copy & paste abstraction (requires finishing new approach to formulas!) - edit before copy to propagate edit to other places (or edit after copy to make it specific to a case) - higher order copying from https://tomasp.net/academic/papers/copy-paste/paint22.pdf * augmenters - cf. bonnie nardi (calls them something else - Jonathan says) - add programming by demonstration data wrangling gui to table (trigger interactions) cf. lorgnette

F Applications

* todo list / counter / maybe too simple * (if used in the walkthrough, maybe something else? board game as in varv - tic tac toe? or 7guis?) * conference organizer * data exploration (ala histogram) * linked charts

G Extras

* metablocks? * self-sustainability * some non-browser implementation of this (as in Varv?)

explicit structure self-sustainability notational freedom

Maybe have 'enabled' for edits afterall? (we can merge with conflicts and disable some edits, but keep them in history for info)

NOTES type Edit = Kind : EditKind Dependencies : Selectors list – only needed for evaluated edits

VALUE vs STRUCTURE distinction * good in theory, nice for implementation * tricky to use! needs some assistance tools

TODO - things to work on * "represent" edits somewhere in document as "library of functions" and then call those from buttons (rather than embedding them directly) allow some kind of abstraction (as in Histogram) to make them reusable * figure out how to do evaluation better (based on the stored abstractions? but need to store provenance...)

SEMANTIC CONDITIONS https://www.youtube.com/watch?v=NBnc2ToS_j0 (has a section on this in background)

SUBSTRATE DESIGN PROBLEMS * selectors - all for structure / index for data (but it is useful to allow others...) (multiselect also bad for checks!) * groups/conditions/preconditions (c.f. email to jonathan) tried conditions on edits; trying groups with check edits * what to do with "disabled edits"? for example when we remove all checked (before, this created edit groups with "check" but if the check was false, the group was ignored and this messed up merging - because we wouldn't know if the edit had any effect or not)

Evaluation * evaluated edits have to be migrated to the end (if there are conflicts, they are dropped) Think of this as maintaining a tree:

e3 | e2 evaluated | / e1 | e0

this has to be serialized as e0 -> e1 -> e2 -> e3 -> evaluated

evaluated edits do not became part of the main history but hang on the side

ISSUES * if we merge a thing with saved-interactions with something, hashes will change!

NOTES * ListAppendFrom - we need this, because we cannot encode this. * for records, we can RecordAdd(sel, fld, ..) @ Copy(sel @ [Field fld], src) but this does not work for lists - because we do not know the index! (and we cannot look into current document, because it will differ for saved-interactions)

TODO * many things with <tag> selectors currently do not work (e.g. 'matches' for highlighting) because if we collect path of a current node, we collect indices and get /some/2/another - and cannot tell if this matches /some/<li>/another - we'd have to collect more detailed path info!

INTERACTION * replay stored event handlers against the old version? (this way, adding an item to a speakers list gets migrated & adds a new table row!) * simlarly!! we need merge in order to apply edits to multiple targets (when you remove all items in a list, the indices change) (but I guess we should do this against version at the time of saving too....)

[this & evaluation = the unreasonable effectiveness of merging]

Notes on storing and reusing edits * references need to be represented as references so that they get updated (NO! not if we reply them against old version, which seems better - but there are 2 design choices) * how to apply them to multiple targets? use Move to update the selectors instead of replacing the prefix manually

IDEA: Type check edit groups to ensure they preserve structure but not individual edits eg when adding list item

CONDITIONALS https://toby.li/files/p311-radensky.pdf

REMAINING IMPLEMENTATION TODOs:

* Some kind of provenance visualization * Some kind of matchers/transformers mechanism (ideally to add interactive buttons to tables) * Apply to all (remove completed in TODO)

append /speakers #hamilton <li> addfld /speakers/#hamilton speaker "Margaret Hamilton, hamilton@mit.edu"

»> wraprec /speakers body <table>

[update selectors - add /body] append /speakers/body #hamilton <li> [dtto] addfld /speakers/body#hamilton speaker "Margaret Hamilton, hamilton@mit.edu"

»> retag /spekers/body <tbody>

[na - out of scope]

»> retag /speakers/body/* <tr>

[apply to added - update node (* => #hamilton scoping)] append /speakers/body #hamilton <li> + retag /speakers/body/#hamilton <tr> [na - out of scope] addfld /speakers/body#hamilton speaker "Margaret Hamilton, hamilton@mit.edu"

»> wraprec /speakers/body/*/speaker value <td>

append /speakers/body #hamilton <li> + retag /speakers/body/#hamilton <tr> + wraprec /speakers/body/#hamilton/speaker value <td> [no op - it is not there] addfld /speakers/body/#hamilton speaker "Margaret Hamilton, hamilton@mit.edu" + wraprec /speakers/body/#hamilton/speaker value <td>

» updid /speakers/body/* speaker=>name

append /speakers/body #hamilton <li> + retag /speakers/body/#hamilton <tr> + wraprec /speakers/body/#hamilton/speaker value <td> [no op - it is not there] + updid /speakers/body/#hamilton speaker=>name [no op - it is not there] addfld /speakers/body #hamilton speaker "Margaret Hamilton, hamilton@mit.edu" + wraprec /speakers/body/#hamilton/spea value <td> + updid /speakers/body/#hamilton speaker=>name

»> addfld /speakers/body/* email ""

append /speakers/body #hamilton <li> + retag /speakers/body/#hamilton <tr> + wraprec /speakers/body/#hamilton/speaker value <td> [no op - it is not there] + updid /speakers/body/#hamilton speaker=>name [no op - it is not there] + addfld /speakers/body/#hamilton email "" [no op - it is not there] addfld /speakers/body #hamilton speaker "Margaret Hamilton, hamilton@mit.edu" + wraprec /speakers/body/#hamilton/spea value <td> + updid /speakers/body/#hamilton speaker=>name + addfld /speakers/body/#hamilton email ""

...

copy /speakers/body/*/name => /speakers/body/*/email edit /speakers/body/*/name (fn1) edit /speakers/body/*/email (fn2)