Baseline: Verson Control All the Way Down

Jonathan Edwards[a] (iD) and Tomas Petricek[b] (iD)

a   Independent, Boston, MA, USA

b   Charles University, Prague, Czechia

Abstract   Action calculus is a technique of change management that generalizes conventional version control and collaborative editing systems to handle refactorings, transformations, schema evolution, and database differencing. We take the first steps of extending it into a new kind of query language. Altogether we solve four of the eight challenge problems of schema evolution identified in a prior paper. The theory is presented semi-formally and a prototype implementation is discussed. A playable demo is submitted as an artifact.

The superpower of the action calculus is transporting actions across alternate timelines while preserving their intention. The *Many-worlds Interpretation of Programming* is a research vision to extend this power into a unified PL/DB/UI, solve the impedance mismatch problem, and integrate facilities for testing, package management, and deployment. Realizing that vision would in effect be a revival of the beloved Lisp/SmallTalk unified programming experience, reimagined for the modern world.

Context: The programming tech stack has inflated to mind-boggling proportions. We must seek ways to simplify programming, if only in certain domains.

Inquiry: Problems of change management recur in many guises throughout programming. Is this a point of attack where a general technique can leverage widespread benefits? We generalize upon prior work in version control and collaborative editing systems.

Approach: In prior work we identified eight challenge problems of schema evolution in programming systems. We developed the action calculus initially to solve two of these. The key move was to shift the focus from states to high-level actions.

Knowledge: We discovered a new technique of change management that works on timelines of high-level actions, not primitive state edits. It offers a much-simplified conceptual model for version control. We iteratively designed a new GUI for visualizing nesting and relationships and differencing them.

Grounding: We solve four of the eight challenge problems of schema evolution identified in prior work. The action calculus has iterated through two prior workshop papers. Demos were used for feedback on the GUI, leading to two complete redesigns.

Importance: Version control and schema evolution are sources of much complaint that deserve improved solutions. More generally we need to research cures for the Complexity Cancer afflicting programming.

Keywords   key, word

The Art, Science, and Engineering of Programming

1　Introduction

We observe that *change managment* is a central problem throughout the theory and practice of programming, albeit appearing in many guises. The goal of this paper is to convert that observation into a research vision of new general purpose techniques for change management that can yield benefits across diverse aspects of programming and software development. To motivate and substantiate this vision we offer a first step: a technique of change managment called *action calculus* that we apply to demonstrate several novel capabilities.

Problems of change management appear in many guises:

- Programming ultimately happens by changing code. There are well-established tools for managing changes to source code but also much discontent with them, indicating a pent-up demand for something better (§7.1).
- Managing state change is a deep concern in the design of programming languages. A central principle of functional programming is to reject mutable state in favor of abstractions like monads and lenses. For their part, imperative languages utilize complex state management libraries. (§7.3)
- Much code is required to translate data changes between a UI, PL data structures, and a DB. It seems like much less code ought to be needed, but extensive research on this *impedance mismatch problem* has not produced a decisive solution (§7.2).
- Collaborative editing and data synchronization engines are concerned with automatically transporting and reconciling changes, applying the techniques of Operational Transformation (OT §7.4) and Convergent Replicated DataTypes (CRDTs §7.5).
- Database schema change necessitates data migration and query rewriting. Live programming faces a similar problem when state becomes stale. There has been much research on *schema evolution* but in practice it is still largely manual and ad hoc (§7.6).
- Tests sometimes simulate input changes, mock output changes, and check for expected changes. Tests sometimes snapshot a system state for convenient comparison, but then can't detect when it is stale.
- Changes in dev must be deployed to prod. Sometimes code changes are wrapped inside feature flags that turn them into runtime changes.
- Changes in upstream dependencies must be installed, often via elaborate and fragile package management systems.

The list could go on. But it is all too easy to notice patterns and draw analogies. Is there some shared essence that can be distilled into a theory and leveraged in a tool to practical benefit? We conjecture yes, though with the caveat that it may be necessary to break compatibility with the established tech stack, at least at first. We believe it would still be a valuable result to show that a unified approach to change management can simplify multiple aspects of programming, even if at first only in a Petri dish. Likewise we argue that performance concerns be deferred.

With that said we propose a new technique of change management called the *action calculus*. Action calculus differs from familiar techniques in that:

1. It starts by defining the data model of a class of artifacts along with a system of actions upon that model. These actions capture high-level design changes such as transformations and refactorings, not just primitive edit operations in the model (or even worse, edits to a textual syntax).

2. The history of actions on an artifact is recorded in a *timeline* by monitoring the API and a GUI that manifests actions as direct manipulations. Because actions know more about the user's intent than primitive edits, an action timeline contains more information than the final state or even the entire history of states.

3. Timelines can be compared to compute the differences between artifacts, expressed as a hypothetical timeline in which a maximally common ancestor was first created and then the two artifacts forked off with minimally diverging timelines. Note that this common ancestor may not have ever actually existed. Differential timelines are more accurate than lower-level techniques because they take into account structural transformations and refactorings.

4. The key feature of the action calculus is the ability to *transport* individual actions across forking timelines while preserving the actions's intent (which is given a precise meaning). We say that transportation navigates the *multiverse* of alternate timelines of an artifact.

Our research conjecture is that the action calculus can be applied beneficially to the varied problems of change management in programming that were listed above, and more. Since the action calculus navigates the multiverse of alternate histories we call our conjecture the Many-worlds Interpretation of Programming.

Realizing that conjecture would in effect be a revival of the beloved Lisp/SmallTalk unified programming experience, reimagined for the modern world. But for now this is a far-off research vision that we offer as the motivation of our work and we hope as an inspiration for others. The concrete contributions of this paper are:

1. We semi-formally define the action calculus on a simple data model. The capabilities of the calculus are explained by example. We simplify the baroque conceptual models of version control into an open world of artifacts with append-only histories where branching is just copying. Transforming between a scalar and a list is used as a point of comparison with other techniques.

2. We extend the simple data model to include relationships, along with actions to handle common schema refactorings in relational databases. The motivating example is normalizing a table by splitting and deduping.

3. We present a GUI design for visualizing nesting and relationships. It also visualizes differences even in the presence of structural transformations. We discuss the design tradeoffs explored in the iterative evolution of this GUI.

4. We submit a *playable demo* as an acccompanying artifact. This demo lets you directly experience the GUI and be guided through the database normalization scenario. We hope that playable demos offer a new way to evaluate HCI research.

5. We take the first steps of extending the action calculus into a query language. Although only a preliminary prototype it demonstrates the key idea that a function can be seen as a hypothetical timeline of imperative actions that extracts the

result from the final state. We show one benefit of this approach: Programming by Demonstration (PbD). Another benefit is that *query rewriting* falls out for free from transporting schema changes. We communicate these very preliminary results because the community may find them intriguing and controversial.

6. Altogether we solve four of the eight challenge problems of schema evolution in a prior paper[2].

2 Simple Structural Action

We introduce action calculus in a simple and familiar setting: nested lists and records. The atomic values are strings, supplied in the set S, and numbers, supplied in N, including `NaN`. What is unusual about this data model is that we assign permanent unique identifiers (IDs) to every record field and list element. These IDs are supplied in the disjoint sets F for record fields and E for list elements. Because record fields have unique IDs their names are only for human readability and we elide in them in most examples. Another unusual feature is that deletion of a record field or list element leaves behind a *tombstone*.

Remark. IDs and tombstones are invisible to both the user and the program. They only affect action transportation (and performance). We believe they are not necessary and have prototyped implementations without them (in fact Baseline currently doesn't use tombstones), but that adds a lot of complexity. We adopt them here in the interest of simplicity.

The model is homogeneously typed similarly to statically typed functional programming (FP) language such as ML, as follows. List elements all have the same type as defined in the type of the list (except tombstones). Record values have the same sequence of fields with the same ID and type of value as in the type of the record. The elements of a list nd the fields of a record must have different IDs. The empty record `{}` serves as a unit type. Every type has an *initial value*. We define the syntax of this model:

type	value	initial value	
$T ::=$	$v ::=$	$T^{\varnothing} =$	
String	S	""	string
Number	N	NaN	number
List T	$[E\colon v \ \ldots]$	$[]$	list
$\{F\ S\colon T\ \ldots\}$	$\{F\colon v\ \ldots\}$	$\{F\colon T^{\varnothing}\ \ldots\}$	record
$\perp$	$\times$	$\times$	tombstone

A *path* is a possibly empty sequence of IDs denoting a path drilling into nested records and lists. Paths can access both values and types. We separate the IDs in a path with dots, and a single dot is the empty (top) path. A special element ID $*$ is used to access the element type of a list type.

A *state* pairs a value with its type. We abuse the notation $v :: T$ to both denote a state and assert that the value matches the type. An *action* is an operation on states, converting an *input state* into a *output state*. Actions are not defined on all input states, imposing various preconditions. All actions take a *target* path as a parameter indicating that the action is to be performed at that path within the state. We define actions as terms of a grammer, using infix syntax, placing the target path first, then the action name, followed by any other parameters. Here are the actions on simple structures:

Value actions	
p noop	Do nothing
p write v	Write atomic value v at p
p insert E before E'	Insert element ID E into list at p in front of element E'
p append E	Append element ID E to end of list at p
p delete E	Delete element ID E in list at p
p moveFrom p'	List element at p is overwritten from p' in the same list or cousin in nested lists, and p' is deleted
Type actions	
p Define T	Define p to have type T, initializing values
p Convert T	Convert p to atomic type T
p Rename S	Rename record field p to string S
p Insert F before F'	Insert field ID F into record at p in front of field F'
p Append F	Append field ID F to end of record at p
p Delete F	Delete field ID F in record at p
p MoveFrom p'	Record field at p is overwritten from p' in the same record or containing/contained record, and p' is deleted
p ListOf	Convert value at p into a list of one element with ID $\mathbb{1}$
p IntoFirst	Convert list at p into its first element, else the initial value
p RecordOf F	Convert value at p into a record of one field with ID F
p IntoField F	Convert record at p into value of its field F

We execute actions with the ⨾ infix operator taking an input state on the left, an action on the right, and yielding an output state which may be chained into subsequent executions. Here is an example that starts with an empty list of numbers and adds an element e with value 1:

$$S_1 = [\,] :: \text{List Number}$$
$$S_2 = S_1 \mathbin{⨾} \,.\ \text{append}\ e \mathbin{⨾}\ e\ \text{write}\ 1$$
$$= [e: 1] :: \text{List Number}$$

The append action targets the empty path `.` referring to the whole list. Then the write action targets the new element with the path `e` which is an abbreviation for `.e`. In all our examples we take IDs like e as given but in practice the Baseline API generates them uniquely.

A *table* is a list of records, the fields of which are the *columns* while the elements of the list are the *rows*. Here is an example of changing the type of a table:

$$S_1 = [e: \{x: 1,\ y: 2\}] :: \mathsf{List}\{x: \mathsf{Number},\ y: \mathsf{Number}\}$$

$$S_2 = S_1 \ \mathring{,}\ * \mathsf{Append}\ z\ \mathring{,}\ *.z\ \mathsf{Define\ Number}$$

$$= [e: \{x: 1,\ y: 2,\ z: \mathsf{NaN}\}] :: \mathsf{List}\{x: \mathsf{Number},\ y: \mathsf{Number},\ z: \mathsf{Number}\}$$

The `Append` action targets the path `*` which is the record type of the list's elements and adds a new field `z` to it, in effect adding a column to the table. The `Define` action targets the path `*.z` which is the type of the new field (initially the unit type {}) and changes it to `Number`. The new field is also inserted into all elements of the list with the initial value `NaN`.

The `append` and `write` actions are *value actions*, meaning they do not modify the type of the state. On the other hand `Append` and `Define` are *type actions* that may modify both the type and value. Type actions do what databases call *schema migration*: the value is adapted to match the type while minimizing loss of information, iterating over list elements as needed (think of the $*$ in the type path as a wildcard). By convention we capitalize the name of type actions. Baseline has a *user mode* in which only value actions are permitted.

Remark. In this presentation we adopt the standard approach of defining values and types as distinct objects. However Baseline only has values, with initial values serving as prototypes. Every list contains an initial *header* element with the ID `*` containing an initial value which serves as the prototype of the list elements. Note how this convention corresponds to the visual rendering of a table where a header row describes the type of each column. Types are a powerful abstraction for the theory and implementation of programming systems but we conjecture that they can be replaced with prototypes, without loss of power, to simplify the programming experience. We have also explored an untyped version of our approach [4].

Many of the type actions exist in order to do schema migration on values: they are not neeeded just to define a type prior to populating it with data. We think of these actions as *type refactorings*, capturing high-level design changes, and manifesting in the GUI as direct manipulations. Schema migration is also a challenging problem for *live programming* [2].

$$S_1 = [e: \{\mathit{what}: \texttt{"clean"},\ \mathit{who}: \texttt{"Jack"}\}] :: \mathsf{List}\{\mathit{what}: \mathsf{String},\ \mathit{who}: \mathsf{String}\}$$

$$S_2 = S_1\ \mathring{,}\ *.\mathit{who}\ \mathsf{ListOf}$$

$$= [e: \{\mathit{what}: \texttt{"clean"},\ \mathit{who}: [\mathbb{1}: \texttt{"Jack"}]\}] :: \mathsf{List}\{\mathit{what}: \mathsf{String},\ \mathit{who}: \mathsf{List\ String}\}$$
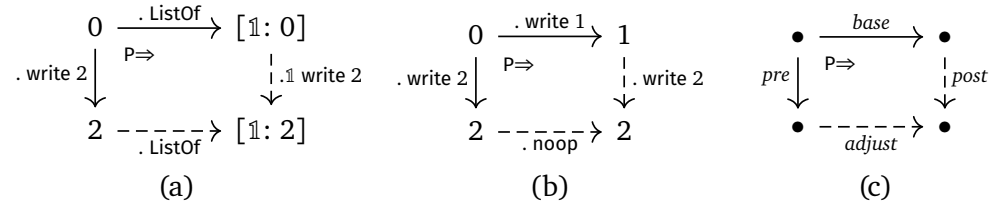
■ **Figure 1** TODO refactoring

For example a common design change is when a single value needs to become a list of multiple values. We capture this refactoring in the `ListOf` action, shown on a

TODO table in Figure 1. The ListOf action takes any type and wraps it inside a list, and wraps all corresponding values into single-element lists using the element ID 𝟙. We will return to this example in the next section.

3 High-fidelity low-fuss version control

In this section we build version control for simple structures that is higher fidelity than standard techniques because it is informed by the history of actions. A *timeline* is an initial state and a sequence of actions executed in order that are each valid on their input state, producing an end state. Baseline records timelines by observing the actions exeuted in the API and the GUI, where actions are manifested as direct-manipulations. We implement version control by comparing and altering timelines. These operations are built upon two primitive functions on timelines: Project and Retract.

3.1 Projection

$$
\begin{array}{ccc}
0 \xrightarrow{\ .\ \text{ListOf}\ } [\mathbb{1}: 0] & \qquad 0 \xrightarrow{\ .\ \text{write 1}\ } 1 & \qquad \bullet \xrightarrow{\ base\ } \bullet \\
\end{array}
$$

$$(a) \qquad\qquad (b) \qquad\qquad (c)$$

■ **Figure 2** Projection

    The Project function takes as inputs two timelines with the same initial state and produces two new timelines continuing from the input timelines and converging on the same final state. It is easier to understand this with diagrams. In Figure 2(a) the arrow on the left takes the state 0 (we elide the obvious type signature) and writes a 2 over it. The arrow on the top wraps that 0 into a list with one element [𝟙: 0]. The projection function takes these two arrows as input and produces the two dashed arrows such that the diagram *commutes*, meaning the dashed arrow actions are valid on the states at their bases, and they both produce the same state [𝟙: 2]. The way this is acheived is to change the target path of the write action to follow the ListOf action. We say that the write actions have the same *intention* – they "do the same thing" – they convert the 0 to a 2, even though the 0 has moved to a different location. That is the key principle of projection: it produce a right-hand action that "does the same thing" as the left-hand action despite the top action having intervened.

    Figure 2(b) shows a messier situation. Here the left and top actions are both writing different values to the same location. This is a *conflict*. The rule is that projection attempts to preserve the intention of the left action, even if that means overriding the top action, which as a consequence gets converted to a noop on the bottom.

    This last example shows that projection is asymmetric: flipping the diagram on its diagonal to switch left and top yields a different result. And in fact we will have

occasion to flip and rotate these diagrams, so we cannot depend on terminology like "left" and "top". Figure 2(c) shows our orientation-neutral terminology: projection converts the *pre* action into the *post* action, preserving its intent after the *base* action intervenes. The *base* action is converted into the *adjust* action preserving as much of its intention as is allowed by the first rule. We also graphically indicate the orientation of the diagram by placing P⇒ into the corner of the initial state.


(a) (b) (c)
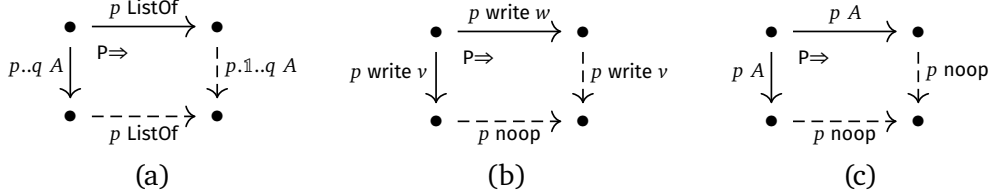
■ **Figure 3** Projection rules

We define projection with a set of rules. Figure 3(a) is a rule covering the ListOf example. The target path of the ListOf is abstracted to p, and the write action is abstracted into any action A targeting p or deeper, which is indicated with the path concatenation operator in $p..q$. Action A is projected down into the ListOf with the path $p.\mathbb{1}..q$.

Figure 3(b) is the rule for conflicting writes. Note that the rule in (c) must override both of the rules (a) and (b) when an action is projected through an identical action. In that case we want them to cancel out into noop actions (see ***).

3.2 Retraction

(a) (b) (c)
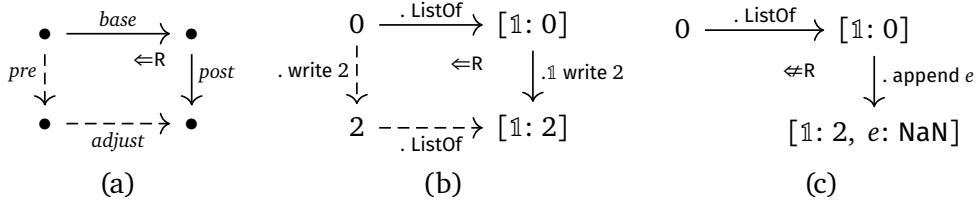
■ **Figure 4** Retraction

The Retract function goes in the opposite direction of Project. Figure 4(a) shows that it starts with two timelines *base* and *post* and produces *pre* and *adjust*. We say that *post* is retracted through *base* into *pre*. The goal as with projection is that *pre* preserves the intention of *post* before *base* happened – it "does the same thing". Likewise *adjust* preserves the intention of *base* but defers to the first rule.

Figure 4(b) shows the retraction reversing the projection in Figure 2(a). Often retraction is the inverse of projection as in this case, but not always, and sometimes retraction is not possible at all. In Figure 4(c) the list has a new element appended to it. There is no way to do the same thing before the list was created. Projection failures indicates a *dependency* between actions. The append action depends on the prior ListOf action having created the list and cannot be retracted through it.

Retraction is defined with rules like those for projection so we skip over that in the interest of brevity.

3.3 Across the multiverse

What are the versions that version control controls? We see them as the result of *branching* timelines: two timelines that branch off of an earlier common state. We can diagram branching like this:

$$A \xleftarrow{a_n} \quad \xleftarrow{\cdots\cdots} \quad \xleftarrow{a_1} O \xrightarrow{b_1} \quad \xrightarrow{\cdots\cdots} \quad \xrightarrow{b_m} B$$

Here the versions are A and B diverging with actions a_i and b_i from a common state O. Our approach to version control is based on the ability to *transport*[1] an action from one branch to another, which we define digrammatically in Figure 5.

$$\text{Transport}(a_{1\ldots n}, O, b_{1\ldots m}) = b' \text{ in}$$

Figure 5 Transportation

Transportation retracts a_n backwards through all earlier actions on the A timeline onto the original state O and then projects that action forwards through all the actions of the B timeline. The final result is the action b' that yields a new version B'. The effect of transportation is that the final action b' preserves the intention of the original action a_n given all the actions traversed traveling backward in time to O and then forward to B. Since transportation uses retraction it may fail, because of a dependency on an earlier action in the A branch. However it is always possible to do a group transport that includes all the transitive dependencies.

As an example we extend the TODO refactoring in Figure 1 into two branches: branch A reassigns the TODO from Jack to Jill; branch B converts the assignment into a list, changes the assignment from Jack to Jacque and inserts a new assignment to Tom:

$O = [e: \{what: \text{"clean"}, who: \text{"Jack"}\}]$

$a_1 = e.who \text{ write "Jill"}$

$A = [e: \{what: \text{"clean"}, who: \text{"Jill"}\}]$

$b_1 = *.who \text{ ListOf}$

$b_2 = e.who.\mathbb{1} \text{ write "Jacque"}$

$b_3 = e.who \text{ insert } g \text{ before } \mathbb{1} \ ; \ e.who.g \text{ write "Tom"}$

$B = [e: \{what: \text{"clean"}, who: [g: \text{"Tom"}, \mathbb{1}: \text{"Jacque"}]\}]$

---

[1] It is well known that a transporter can access alternate timelines[5]

Baseline: Verson Control All the Way Down

If we transport b_2 into A the change from Jack to Jacque gets applied to the who field (showing the change in bold):

$A' = [e: \{what: \text{"clean"}, who: \textbf{"Jacque"}\}]$. If we transport a_1 into B the change from Jack to Jill gets applied to the list element containing Jack, even though it has been wrapped and shifted:

$B' = [e: \{what: \text{"clean"}, who: [g: \text{"Tom"}, \mathbb{1}: \textbf{"Jill"}]\}]$.

   This example demonstrates how transportation can propagate changes bidirectionally through structural changes. We offer this as a solution to challenge problem #7 in Edwards et al. [2], where no plausible state invariant is found able to generate the expected behavior. Our solution is that action histories let us track how locations shift through time in a way that cannot be inferred just by comparing states – in effect the IDs we add to the state are a cache of this historical information.

3.4 Correctness

How do we know if the rules for projection and retraction are correct, that they do indeed preserve the intentions of actions? Our answer is that there is no objective answer to this question. The rules are essentially axioms defining what it means to preserve intention. They may turn out to be more or less in agreement with intuition. We have learned that these rules are closely coupled to the semantics of the actions and offer many alternative design choices.

   Nevertheless there are some principles we can expect any system of transportation rules to follow. One of them is that projection and retraction should be inverses – that transporting an action to a different branch and then back again should return it unchanged. Doing the same thing should be a symmetric relation. Unfortunately that principle is violated by the examples we have already seen where an action gets wiped out during transport into a noop, which will always return as a noop. We need to weaken the principle to say that roundtripping an action must return an action that does the same thing or less than the original. Accordingly we define an ordering on branches $\leq$:

$$O;a \leq O;b \quad \text{iff} \quad \begin{array}{ccc} O & \xrightarrow{\ b\ } & B \\ a\downarrow & \text{P}\Rightarrow & \vdots \\ \downarrow & & \downarrow \\ A & \dashrightarrow & B \end{array}$$

in other words, projection produces a fixpoint. The **Roundtrip Monotonicity** principle asserts that (a) $\leq$ is a partial order, and (b) roundtripping a timeline through a projection and retraction or the reverse results in a timeline $\leq$ the original.

3.5 Branch minimization and synthesis

We have been discussing transportation assuming we are given branching timelines. But these might be redundant: there could be changes duplicated in both branches, either by accident or by explicit transportation. There could also be redundant changes within a branch that override earlier ones. Such redundancies are undesireable because they add noise to a visualization of differences, and in some cases can cause

transportation to produce surprising results. Luckily transportation itself can be used in a simple algorithm to minimize branches.

Note that in Figure 5 transportation produces on the bottom of the diagram two new timelines $adjust_a$ and $adjust_b$ branching from a new shared state O'. These *adjusted* timelines eliminate the redundancy of the original action and its transported image. Clearly after executing a transportation we should drop the original branching timelines and use the adjusted ones instead. We can exploit this side-effect of transportation to minimize branching timelines by removing all redundancies. Given a pair of branching timelines we iterate over all the actions on both sides in time order. For each action we try to transport it to the other branch. If the transportation succeeds and its result is a fixpoint (it doesn't alter the state of the other side), then we discard the branches in favor of the adjusted ones and continue the loop. The intuition here is that if the action makes no difference when transported to the other side then it isn't really a difference, and we only want to include actual differences in the branching timelines. Therefore we remove it and apply it to the shared state.

Branch minimization means we can synthesize optimal branching timelines from the action histories of any two states. A complete action history always starts from the unit value as its initial state. So all complete histories are branches from the initial state. GIven any two histories we can apply the branch minimization algorithm to synthesize optimal branching timelines for their final states. It is optimal in the sensse that the history of the shared state is maximal while the timelines branching off of it are minimal.

3.6 Low-fuss version control

The ability to synthesize optimal branching timelines for any two states lets us simplify version control by eliminating the *repo*. In conventional version control systems the repo is a database recording versions (typically batched into user-initiated *commits*), and tracking the history of branching and mergeing operations, forming a DAG of commits. The differences between two commits are calculated with *3-way differencing* which depends upon finding thire last common ancestor in the DAG. As we have pointed out this can be imprecise. But the bigger problem is that the repo imposes a large burden of management and training. In particular git is infamous for causing confusion and frustration [1, 3].

We propose a simpler model: an artifact (database, file, directory tree, etc.) under version control will record its action history internally or in an attachment (or only store the history and derive/cache the state). These histories are append-only. To branch an alternate version of an artifact, just make a copy, which will include the history up to that moment. The copies can then diverge. That's it: branches are copies. We don't need to record branching events in a repo, because we can synthesize it on need by comparing the artifact histories, and we can do so more precisely than the historical branching DAG. Instead of a confusion of operations like merge, rebase, and cherry-pick there is the single operation of transportation which can be grouped into larger bulk operations.

Baseline: Verson Control All the Way Down

    Eliminating the repo has another benefit beyond simplicity: artifacts no longer need to belong to a repo. Artifacts are self-sufficient entities in an open world participating in overlapping and shifting webs of collaboration beyond the boundaries of a repo. We essentially return to good old file sharing, which has withstood the test of time. Nevertheless it must be admitted that repos and their tool ecosystems provide benefits lacking in our proposal, especially for large projects and large teams. However the possibility that a conceptually simpler and lighter-weight approach could benefit many less-intensive contexts deserves investigation.

Jonathan Edwards and Tomas Petricek

Baseline: Verson Control All the Way Down

3.7 OT

3.8 Deletion

3.9 much ado about nothing

3.10 undo

4 Relationships and normalization

5 Visualizing and manipulating the multiverse

5.1 Exploring structure

5.2 Finding commonality

5.3 Design tradeoffs

5.4 A playable demo

6 Queries as hypothetical timelines

6.1 PbD

6.2 Query rewriting via projection

7 Related work

7.1 Version control

7.2 The impedance mismatch problem

7.3 PL state management

7.4 OT

7.5 CRDT

7.6 Schema Evolution

8 Discussion

9 Commencement

References

[1]   Luke Church, Emma Söderberg, and Elayabharath Elango. "A case of computational thinking: The subtle effect of hidden dependencies on the user experience of version control." In: *PPIG*. 2014, page 16.

[2] Jonathan Edwards, Tomas Petricek, Tijs van der Storm, and Geoffrey Litt. "Schema Evolution in Interactive Programming Systems". In: *The Art, Science, and Engineering of Programming* 9.1 (Oct. 2024). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2025/9/2. URL: http://dx.doi.org/10.22152/programming-journal.org/2025/9/2.

[3] Santiago Perez De Rosso and Daniel Jackson. "What's wrong with git? a conceptual design analysis". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pages 37–52. ISBN: 9781450324724. DOI: 10.1145/2509578.2509584. URL: https://doi.org/10.1145/2509578.2509584.

[4] Tomas Petricek and Jonathan Edwards. "Denicek: Computational Substrate for Document-Oriented End-User Programming". In: UIST '25. New York, NY, USA: Association for Computing Machinery, 2025.

[5] Wikipedia contributors. *Mirror, Mirror (Star Trek: The Original Series) — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Mirror,_Mirror_(Star_Trek:_The_Original_Series)&oldid=1294096972. [Online; accessed 23-August-2025]. 2025.

About the authors

Jonathan Edwards is an independent researcher interested in simplifying and democratising programming by collapsing the tech stack. He is known for his Subtext programming language experiments and his blog at alarmingdevelopment.org. He has been a researcher at MIT CSAIL and CDG/HARC. He tweets @jonathoda and can be reached at jonathanmedwards@gmail.com.
https://orcid.org/0000-0003-1958-7967

Tomas Petricek is an assistant professor at Charles University. He is interested in finding easier and more accessible ways of thinking about programming. To do so, he combines technical work on programming systems and tools with research into history and philosophy of science. His work can be found at tomasp.net and he can be reached at tomas@tomasp.net.
https://orcid.org/0000-0002-7242-2208