

The Many-worlds Interpretation of Programming

Jonathan Edwards^a  and Tomas Petricek^b 

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. A fragment of a new kind of query language is included. Altogether we solve 4/8 of the 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 the action calculus 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 live programming experience in a single unifying language, 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 4/8 of the 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

Perspective The Art of Programming

Area of Submission Database programming, Visual and live programming, Programming environments, Version control



© Jonathan Edwards and Tomas Petricek
This work is licensed under a “CC BY 4.0” license
Submitted to *The Art, Science, and Engineering of Programming*.

1 Introduction

We observe that *change management* 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 management 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 explores the *multiverse* of alternate timelines of an artifact.¹

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 others not listed). Since the action calculus explores the multiverse of alternate histories we call our conjecture the Many-worlds Interpretation of Programming.

Maybe too cute. Alternate titles:

Action Calculus: Programming in the Multiverse

Action Calculus: Managing the Multiverse

Programming in the Multiverse

Managing the Multiverse

Realizing that conjecture would in effect be a revival of the classic Lisp/SmallTalk live programming experience in a single unifying language, 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.

¹ It is well known that a transporter can access and create alternate timelines[3]

The Many-worlds Interpretation of Programming

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 accompanying 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 present the first steps towards 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 4/8 of the challenge problems of schema evolution in a prior paper[1].

2 Simple Historical Structures

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. There are two special list element IDs: $*$ and \perp . Another unusual feature is that deletion of a record field or list element leaves behind a *tombstone*. IDs and tombstones are invisible to both the user and the program – they record historical information utilized by the action calculus. We discuss this design decision in ***.

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 and 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	
$T ::=$	$v ::=$	$T^\emptyset =$	
String	S	$""$	string
Number	N	NaN	number
List T	$[E : v \dots]$	$[]$	list
$\{F : T \dots\}$	$\{F : v \dots\}$	$\{F : T^\emptyset \dots\}$	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 construct paths with slashes as in Unix file paths. The 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 a *pre-state* into a *post-state*. All actions take a *target* path as a parameter indicating that the action is to be performed at that path within the state. For example the write action takes a value as another parameter and writes that value into the target path. Actions are not defined for all pre-states. The write action requires that the target path exists in the value and that its type matches the value parameter. We define actions as terms of a grammar, using infix syntax, placing the target path first, then the action name, followed by any other parameters, like this: `p write v`. Here are the actions on simple structures:

Value actions	
<code>p write v</code>	Write atomic value v at p
<code>p insert E before E'</code>	Insert element ID E into list at p in front of element E'
<code>p insert E</code>	Append element ID E to end of list at p
<code>p delete E</code>	Delete element ID E in list at p
<code>p moveFrom p'</code>	List element at p is overwritten from p' in same list or cousin in nested lists, and p' is deleted
Type actions	(Iterated over all values paths matching type path p)
<code>p Define T</code>	Define p to have type T , initializing values
<code>p Convert T</code>	Convert p to atomic type T
<code>p Insert F before F'</code>	Insert field ID F into record at p in front of field F'
<code>p Insert F</code>	Append field ID F to end of record at p
<code>p Delete F</code>	Delete field ID F in record at p
<code>p MoveFrom p'</code>	Record field at p is overwritten from p' in same record or containing/contained record, and p' is deleted
<code>p ListOf</code>	Convert value at p into a singleton list with element ID 1
<code>p FirstOf</code>	Convert list at p into its first element, else the initial value

We execute actions with the `;` infix operator taking a pre-state on the left, an action on the right, and yielding a post-state which may be chained into subsequent executions. Here is a concrete example defining a record type `Name` given field IDs `first` and `last`:

The Many-worlds Interpretation of Programming

$$\begin{aligned} \text{Name} &= \{\text{first} : \text{String}, \text{last} : \text{String}\} \\ s_1 &= \{\text{first} : \text{"John"}, \text{last} : \text{"Smith"}\} :: \text{Name} \\ s_2 &= s_1 \circ \text{last write "Smythe"} \\ &= \{\text{first} : \text{"John"}, \text{last} : \text{"Smythe"}\} :: \text{Name} \end{aligned}$$

The write action is a *value action*, meaning it does not modify the type of the state. On the other hand *type actions* may modify both the type and value. By convention we capitalize the name of type actions. Our system has a *user mode* in which only value actions are permitted.

Remark. In this presentation we have adopted the conventional approach of defining values and types as distinct objects. However our implementation 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 action calculus[2], but that is not the topic of this paper.

The Define action is the type-level equivalent of write: `p Define T`. The path p must exist in the state's type, in which case it is changed into the new type T . Any path in the state's value that *corresponds* to the path p is changed into the initial value of the new type T^\emptyset . A value path q correspond to a type path p iff replacing every list element ID in q with `*` makes it equal to p . In other words, the initial value is written into all list elements corresponding to a list type. For example:

$$\begin{aligned} s_1 &= [e_1 : 1, e_2 : 2] :: \text{List Number} \\ s_2 &= s_1 \circ * \text{Define String} \\ &= [e_1 : "", e_2 : ""] :: \text{List String} \end{aligned}$$

The Define action is typically used to define new types. To change the type of populated data as in the above example the Convert action is more useful. For example:

$$\begin{aligned} s_1 &= [e_1 : 1, e_2 : 2] :: \text{List Number} \\ s_2 &= s_1 \circ * \text{Convert String} \\ &= [e_1 : "1", e_2 : "2"] :: \text{List String} \end{aligned}$$

What happens if we convert a string into a number but the string doesn't parse as a number? There are many design choices like this when defining a system of actions, with different tradeoffs. One might choose to yield NaN. Our implementation instead wraps the string in an error value that can be repaired later in the UI.

The Convert action can only convert between atomic types like strings and numbers. Structural transformations are handled with different actions, which is where things get interesting.

3 Transportation

Transport by example: pluralize, insert

3.1 Projection and retraction

3.2 Branchless diffing

Conceptual model Algorithm

4 Relationships and normalization

5 Visualizing the multiverse

5.1 Exploring structure

5.2 Finding commonality

hard problem is seeing what *hasn't* changed, just moved

5.3 Design tradeoffs

Ultorg. iterations on UI

The Many-worlds Interpretation of Programming

5.4 A playable demo

6 Queries as hypothetical timelines

6.1 PbD

6.2 Query rewriting via projection

7 Related work

actually related work: Alex's worlds. Roly. Ask them for comment? Challenge problem paper. I&S: cambria, peritext, universal version control Braid time machine <https://braid.org/time-machines>

7.1 Version control

git, darcs, pijul

git is the most established tool for change management but is also the source of much discontent. Pull refs from schema change paper. santiago's stuff. intentional?

7.2 The impedance mismatch problem

orthogonal persistence, ORMs, Linq, single tier (Tier, hop), naked objects, DBOS, durable computation

7.3 PL state management

FP: Monads, Lenses

Imperative: immutable, React state management

7.4 OT

7.5 CRDT

7.6 Schema Evolution

just ref challenge problems paper?

8 Discussion

pros and cons

caveats, weaknesses, open problems

9 Commencement

summary and call to action


References

- [1] 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>.
- [2] 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.
- [3] 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](https://en.wikipedia.org/w/index.php?title=Mirror,_Mirror_(Star_Trek:_The_Original_Series)&oldid=1294096972). [Online; accessed 23-August-2025]. 2025.


The Many-worlds Interpretation of Programming

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>