Schema Change: Challenge Problems

Jonathan Edwards[a], Tomas Petricek[b], Tijs van der Storm[c,d], and Geoffrey Litt[e]

a Independent, Boston, MA, USA

b Charles University, Prague, Czechia

c Centrum Wiskunde & Informatica (CWI), Amsterdam, Netherlands

d University of Groningen, Groningen, Netherlands

e Ink & Switch, Planet Earth

Abstract Schema change is an unsolved problem in both live programming and local-first software. We include in schema change any change to the expected shape of data, whether that is expressed explicitly in a database schema or type system, or whether those expectations are implicit in the behavior of the code.

Schema changes during live programming can create a mismatch between the code and data in the running environment. Correspondingly, Schema changes in local-first programming can create mismatches between data in different replicas, and between data in a replica and the code colocated with it. In all of these situations the problem of schema change is to migrate or translate existing data in coordination with changes to the code.

This paper contributes a set of concrete scenarios involving schema change that are offered as challenge problems to the live programming and local-first communities. We hope that these problems will spur progress by providing concrete objectives and a basis for comparing alternative solutions.

Context: *What is the broad context of the work? What is the importance of the general research area?* Schema change is an unsolved problem in live and local-first programming that is poorly understood and calls for a more fundamental analysis. It is particularly important as more programming is done in programming systems where programs may be changed by their users and as more programming is local-first, meaning that such changes need to be synchronized.
Inquiry: *What problem or question does the paper address? How has this problem or question been addressed?* Specific systems have specific solutions, but what is lacking is a general framework for thinking about schema change and also a good suite of challenge problems to motivate and evaluate new solutions.
Approach: *What was done that unveiled new knowledge?* We collect challenge problems from a range of different kinds of (primarily) live and local-first application contexts. We develop a framework for thinking about program change and use it to present our challenge problems in a systematic way.
Knowledge: *What new facts were uncovered? What new capabilities are enabled by the work?* We understand change in programs in a novel way, using a two-dimensional framwork that looks at the different pace layers in a program and at the different ways in which change is propagated across program variants.
Grounding: *What argument, feasibility proof, artifacts, or results and evaluation support this work?* We use the framework to provide a framing for a range of challenge problems derived from both existing literature and our own expertise.
Importance: *Why does this work matter?* This paper contributes a set of concrete scenarios involving schema change that are offered as challenge problems to the live programming and local-first communities. We hope that these problems will spur progress by providing concrete objectives and a basis for comparing alternative solutions.

Keywords schema change, database migration, live programming, local-first programming

The Art, Science, and Engineering of Programming

Perspective The Engineering of Programming

Area of Submission Programming Systems

© J. Edwards, T. Petricek, T. van der Storm, G. Litt
This work is licensed under a “CC BY 4.0” license
Submitted to *The Art, Science, and Engineering of Programming*.

1 [TVDS] Super rough attempt at the paragraph

Modern day software development is collaborative. Single user programming does not exist anymore. Just as document editing is becoming more collaborative through online systems like pioneered by Google Docs, programming would benefit form online collaboration. In a sense this is live programming++: whereas traditional live programming systems assume a single developer editing code and see their changes immediately reflected in the running system, online collaborative programming is the same but for multiple programmers at the same time. The key challenge here, we posit, the *schema change*. We take inspiration from database migration, model evolution, and version control, and identify benchmark problems. This is already evident in live programming: editing a program, consisting of type definitions defining the structure run-time state and (transient) procedures, functions etc. operating on that state, requires run-time state to be migrated, after the programmer edits their program. Adding the dimension of multiple programs brings us into the domain local-first, and decentralized version control, where copies are created and edited, which eventually have to be reconciled again (e.g., through merging). Whereas schema evolution and its associated migration problems has seen a lot of attention in the database world, the model-driven world, and the bidirectional transformation (BX) communities, the techniques are still too low-level, too manual, and too much focused on a single user perspectives. By identifying the similarities between analogous problems in live, multi-user programming systems, we hope to stimulate further research on more comprehensive, automatic, high-level, and online solutions. So that the benefits of live programming can be enjoyed in a global and distributed setting involving multiple programmers at the same time.

2 [JE] Type evolution for live programming

Many past improvements in the practice of programming worked by speeding up feedback loops. Live Programming aspires to the optimum of immediate feedback. The problem is that modern software is built out of many layers of different technologies – thoroughly live programming would propagate changes immediately up and down the entire stack of a running system, from the database to the code, and from the client to the server. We call this *full-stack* live programming. Another problem is that modern software is built with complex collaborations involving development, testing, deployment, and suppport. Such workflows are themselves feedback loops – thoroughly live programming would propagate changes between participants frictionlessly. We call this *end-to-end* live programming.

We propose a research agenda towards this greater vision of live programming. The starting point is database *schema evolution*, which propagates database schema changes into data migrations and query rewriting. We call for generalized *type evolution* that works immediately across the entire software stack, and automatically across the entire software workflow. This paper contributes a set of challenge problems to

serve as targets for research on type evolution and as a basis of comparison for the results.

3 [GL] Rough paragraph

Most programming languages analysis treats state as ephemeral: tied to the lifetime of a running program, and reset when the program is edited. But when we take a broader view, most *programming systems* (cite) have state that lasts beyond the lifetime of a single execution. This includes collaboration applications with state persisted in a database, image-based systems like Smalltalk, and live programming environments where the program can be edited without resetting all state.

Although these programming systems exist in very different contexts, they all encounter similar problems related to the essential difficulty of preserving state across program edits. New versions of the program must correctly handle pre-existing data. When the data schema or type definitions change, existing data must be evolved to conform to the new schema. Changing data representations or types can also require changing corresponding parts of the program. These problems have been noticed, named, and studied in these different programming systems: “schema evolution” in the databases literature, “hot reloading” in live programming, etc.

In this paper, we draw connections between these different contexts, pointing out essential shared structure in the hopes of motivating further research on this important problem and encouraging knowledge sharing across subfields. Our contributions are: 1) a shared framework for thinking about the relationship between schema, code, and data in a programming system, 2) Case studies of the framework in a highly diverse set of programming systems, motivated by real-world examples.

4 [TP] Rough paragraph

Programming is increasingly collaborative and interactive. We welcome this trend, because greater collaboration and interactivity make programming more effective. Future programming environments enable programmers to collaborate at a more fine-grained level, adopting individual code changes created by their collaborators as needed. They also let programmers interactively edit code, often during program execution.

In order to support this new way of working, programming systems increasingly have to tackle the problem of schema evolution. If a programmer imports a change that edits a schema, code and data that relies on such schema needs to be adapted to match the new schema. Similarly, if a programmer in a live programming environment modifies their type definition, current program state needs to be updated to match the new structure of types. This is a well-understood and well-studied problem in database community, but it has received little attention in work on programming systems.

The aim of this paper is to provide a conceptual framework for talking about schema evolution in collaborative and live programming systems. Taking inspiration from Stewart Brand's analysis of pace layers, we propose to think of the problem as the problem of evolution of occuring at three layers, the most permanent schema layer, less permanent code layer and least permanent data layer. A change at the most permanent layer affects the less permanent layer, which then has to be updated to keep correspondence between the layers.

We consider four case studies of such schema evolution in collaborative and live programming systems. We present those throught the perspective of our unified conceptual framework. The work both sheds a new light at how schema change needs to be handled in programming systems and also provides a range of challenges that show interesting problems that future programming systems will need to tackle.

5 Introduction

[JE] We should adopt the term Schema Evolution, which is favored in the research literature, defined to include schema change, data migration, and query rewriting.

Schema change won't go away. Changing requirements and changing code lead to changing the schema of a database. Schema change, also called schema migration, is the problem of migrating existing data from the old schema to the new. This often involves custom migration programs or specialized Domain Specific Langiuages. The migration must be carefully coordinated with upgrading the application code and associated artifacts that assume the new schema. Schema change on database servers is often delegated to Database Administrators and DevOps. Live programming [**tanimoto90**, **Hancock03**] and local-first software [**localfirst**] both move data away from the server, eliminating those jobs, but not eliminating the need to do schema change, and indeed increasing the need to do it automatically. Recently **Cambria** [**Cambria**] spotlighted these problems and proposed an approach using lenses [**Foster2007**], but otherwise there has been surprisingly little research. To promote further progress we offer a set of challenge problems to the live programming and local-first communities, inviting them to propose and compare solutions.

Live programming seeks to erase the boundary between editing and running programs. In order to do so program data must be kept around while the program is being edited. Classic Lisp and Smalltalk systems integrated code and data into a persistent *image* [**Sandewall78**, **Goldberg80**]. An interactive shell or *Read Eval Print Loop (REPL)* [**Deutsch64**] is more transient than an image, but still builds up a context of data over long-lived programming sessions the loss of which disrupts the programmer's workflow. In all of these environments programs eventually get changed to create and expect data in a form incompatible with extant data. This happens whether or not the form of the data is specified explicitly in a type system or database schema, or whether it is left implicit in the behavior of the code. Some languages can tolerate a larger set of such changes, most notably Smalltalk which will automatically insert and delete

members in existing instances when a class definition is changed [**Goldberg80**].[1] But there is still a large class of changes that create incompatibilities with existing data, whether it is inside an image, programming environment, REPL, or an external database. Some live programming environments generate data with unit tests, but that only shifts the problem of schema change to adapting those tests. One way or another, stopping everything to manually deal with schema change contradicts the goals of live programming. Schema change won't go away.

Local-first software faces related problems. Its goal is to empower users by moving code and data from the cloud to the user's own devices. Distributed programming techniques like Convergent Replicated Data Types (CRDTs) [**Shapiro11**] are used to coordinate data changes peer-to-peer. Unfortunately these techniques so far have not addressed schema change nor code deployment The traditional techniques of schema change used in centrally managed databases are complicated by the distributed and intermittently connected nature of local-first data. Schema change won't go away.

In the following sections we present a series of challenge problems dealing with schema change in the context of live programming and local-first software. These problems are necessarily expressed using established idioms or conventions, but nevertheless we welcome solutions that translate the spirit of the problem into other contexts.

TODO: Need to explain that we take the programming system view - program is not something built centrally, but something that exists and can be modified

TODO: Maybe the right way to frame this is to say that the challenges are intended less for evaluating particular solutions but more for framing and making sense of different kinds of schema-change-related problems. You can read this paper and when you face a schema-change-related problem, think about it in terms of our dimensions. Maybe compare it to our challenges. We do not necessarilly expect that people will directly solve our challenges though. (Although some might and that would be nice..)

TODO: I also guess this is more about just "change" than "schema change" which is one particular (most challenging) instance - i.e., change at the most permanent pace layer.

[1] Gemstone turns the Smalltalk image into a production-quality database and accordingly provides a sophisticated schema change API [**Gemstone**]. Schema change won't go away.

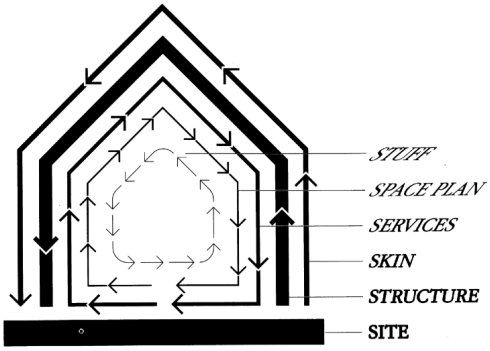

Figure 1 Pace layers from Stewart Brand's classic *How Buildings Learn* [**Brand95**]. The outer layers rarely change, while the inner layers change frequently. The geographic site of a building is eternal, its structure remains stable for decades, space plan needs to change every couple of years. "Because of the different rates of change of its components, a building is always tearing itself apart." [JE] Except the outer skin changes faster than the structure.

6 Dimensions of change

[JE] 'change in a program' might be misunderstood as changes to the source code. Perhaps 'system' or 'software system' might be better than 'program'?

In the broadest sense, this paper is concerned with the difficulties posed by a change in a program. The first dimension of difficulties that we consider in this paper is local. When something in a program changes, other parts of the program typically need to be brought to sync with the changed part. The complexity of the problem varies. If nothing else in the program depends on the changed part, for example when changing a runtime value of an object field, the change poses no difficulty. If many other parts of the program depend on the changed part, the difficulty is greater. For example, if we change the schema of data structures used in the program, we also need to change code that works with it and existing data that the users created. To structure our discussion about this part of the problem, we will use the idea of pace layers introduced by Brand [**Brand95**, **Brand18**] (Figure 1).

The second dimension that we consider in this paper is non-local. A program may have multiple variants that exist independently and need to be synchronized. The difficulty of the problem depends on what kind of synchronization we want to support. In the easier case, all variants of the program converge, i.e. all variants should eventually adopt all changes. In the more challenging case, variants of a program may diverge, i.e. a user may want to apply only certain changes from other users. Note the non-local dimension may interact with the pace layers of the local dimension. A typical case may be that the code and schema of a program should converge, while data and the transient state of a program may diverge.

Dimensions. The framework for talking about schema change used in this paper thus distinguishes between two dimensions:

- The *local dimension* is concerned with difficulties that a change causes across different program pace layers that exist within a single program.
- The *non-local dimension* is concerned with the difficulties that a change causes across different program variants that exist independently.

Program pace layers. As with buildings (Figure 1), programs consist of multiple pace layers that change at a different rate. As with buildings, making a change to a more permanent layer also typically affects all the less permanent layers. Making a change at a more permanent layer is thus more challenging. Alas, changing more permanent structures in programming is frequently necessary. In this paper, we distinguish between four layers.

- *State* – transient state of a program. This is typically never shared across multiple variants of a program and it can often be discarded when a change occurs.
- *Data* – data collected by the program. This represents a more permanent structure that is typically stored in a database and outlives program restarts.
- *Code* – program logic. This layer changes whenever any aspect of the program logic is modified. A change typically affects transient state, but may not affect data.
- *Schema* – the structures of program data. A change at this layer typically requires adapting code to work with the new structure and also updating data and state.

Program variants. We are interested in programs that are distributed across multiple environments as, for example, in local-first software. As discussed earlier, one aspect of this dimension is whether all variants of a program eventually converge or whether it is possible to maintain a diverging version. It is also useful to distinguish the directions in which a change may flow, i.e., can a change originate in any variant or is there a single centralized source of change. Note that each of this characteristics may be associated with individual pace layers of a program.

- *Convergence* vs. *Divergence*. In the convergence model, all program variants eventually adopt all changes. It may not be possible to adopt a particular change before adopting other changes it depends on. In the divergence model, a user may choose only particular changes they want to adopt.
- *Centralized* vs. *Decentralized*. In the centralized model, changes (at a specific layer) can only originate from a particular source. In the decentralized model, changes can be done on any of the multiple co-existing variants of a program.

Challenge problems. The framework outlined above offers a wide range of configurations and one could conceive of a challenge problem for many of those configurations. We can start with a change at each of the four layers, consider each of the four configurations with respect to program variants (and also consider different behaviour at all the affected less permanent layers). The aim of this paper is, however, not to enumerate all options. Instead, our goal is to present a number of typical and often encountered challenge problems. We use the above framework to more precisely define the nature of the challenge problem. We also often use the above framework to discuss variations on the particular challenge.

7 Divergence Control

This section was an extension of the Extract Entity challenge but should be lifted up to the top-level discussion of divergence

7.1 Context

Schema evolution becomes more challenging when data, code, and schema cannot all be updated together in an atomic way. In these situations, different versions must coexist and interoperate with one another: for example, edits made in one schema must be applied in another, or code written against one schema must run against data stored in another.

Such situations are common in software engineering. Web backend architectures often put data and code on separate machines, which makes it impossible to update both atomically. To perform upgrades, teams must manually perform multi-step [zero-downtime deployment processes](https://planetscale.com/blog/zero-downtime-rails-migrations-planetscale-rails-gem) to step the database and the code servers forward gradually such that they are compatible with one another at any given step. One example would be deploying a change to add a database column before deploying the code that uses the column.

In the simplest cases, divergence can last only for a brief moments in between zero-downtime deployment steps. But divergence can also occur across longer timescales, especially in situations where the different parts of a system are not all centrally controlled. For example, a public web API may need to support applications using old versions of the API for months or years, since it is not realistic to force all consumers to upgrade immediately. Divergence can even be indefinite – **Cambria** [**Cambria**] addresses collaborative applications where different users may want to collaborate on shared data through different tools using different schemas, with no intention of ever converging.

Divergence does not just occur in software engineering; it also appears in end-user workflows. Users frequently make copies of structured documents like spreadsheets and then diverge them by altering both data content and schema. In the case of spreadsheets schema changes include rearrangements to the structure of rows and columns as well as changes to formulas. The problem is that the users want to transfer such data and schema changes between these divergent copies, and they want to pick and choose which of these changes to transfer. In practice such transfer is done manually through copy & paste. **Basman19** [**Basman19**] has documented an ecology of emailed spreadsheets. **Burnett14** [**Burnett14**] distilled field observations of these practices in the story of Frieda, which we quote here:

> For example, consider “Frieda”, an office manager in charge of her department’s budget tracking. (Frieda was a participant in a set of interviews with spreadsheet users that the first author conducted. Frieda is not her real name.) Every year, the company she works for produces an updated budget tracking spreadsheet with the newest reporting requirements embedded in its structure and formulas. But this spreadsheet is not a perfect fit to the kinds of projects and

sub-budgets she manages, so every year Frieda needs to change it. She does this by working with four variants of the spreadsheet at once: the one the company sent out last year (we will call it Official-lastYear), the one she derived from that one to fit her department's needs (Dept-lastYear), the one the company sent out this year (Official-thisYear), and the one she is trying to put together for this year (Dept-thisYear).

Using these four variants, Frieda exploratively mixes reverse engineering, reuse, programming, testing, and debugging, mostly by trial-and-error. She begins this process by reminding herself of ways she changed last year's by reverse engineering a few of the differences between Official-lastYear and Dept-lastYear. She then looks at the same portions of Official-thisYear to see if those same changes can easily be made, given her department's current needs.

She can reuse some of these same changes this year, but copying them into Dept-thisYear is troublesome, with some of the formulas automatically adjusting themselves to refer to Dept-lastYear. She patches these up (if she notices them), then tries out some new columns or sections of Dept-thisYear to reflect her new projects. She mixes in "testing" along the way by entering some of the budget values for this year and eyeballing the values that come out, then debugs if she notices something amiss. At some point, she moves on to another set of related columns, repeating the cycle for these. Frieda has learned over the years to save some of her spreadsheet variants along the way (using a different filename for each), because she might decide that the way she did some of her changes was a bad idea, and she wants to revert to try a different way she had started before.

7.2 Example

The Divergence Control challenge instantiates the problem of divergent state with the example from the Extract Entity challenge. Every month the orders department sends its spreadsheet to the accounting department, which adds the data to a version of the spreadsheet that it has customized for financial tracking. But when the orders department migrates its spreadsheet to extract out customers the accounting department does not want to conform. They could manually convert incoming data in the new schema into their variant of the old schema. But they shouldn't have to. It should be possible to run new data through the schema change in reverse, converting it back into the format the accounting department is used to ingesting. Note that this is not a matter of synchronizing the divergent spreadsheets – they maintain differently evolved schema.

7.3 Challenge

What is needed is a user-friendly mechanism for transferring data changes from one schema to another bidirectionally. In one direction we want to transfer changes made in the new schema into a variant of the old schema. The opposite direction might be needed, for example, if the accounting department makes corrections to

customer addresses, which ought to be pushed through into the orders department's new schema.

Bidirectional transfer of changes between divergent copies is similar in some ways to source code version control as in Git [**ProGit**]. There is *forking* of long-lived divergent copies. We want to *diff* these forks to see exactly how they have diverged. We want to partially *merge* them by *cherry picking* certain differences. Yet there are also many dissimilarities with Git: our data is more richly structured than lines of text; our schema changes are higher-level transformations on these structures than inserting and deleting characters; and we expect that end-users be able to understand it [**gitless**]. The Divergence Control challenge is in a sense to provide "version control for schema change" meeting these criteria, with the key technical challenge being the ability to transfer selected differences bidirectionally through schema changes as independently as possible.

7.4 Goals

This challenge invites a change of perspective in both live programming and local-first software. Live programming must move from being a solitary activity to a collaborative workflow, and one where the collaboration is not just on editing source code but on live integrated code and data, as in the Smalltalk/Lisp images of old.[2] For its part, local-first software must move from automatically converging data replicas to also interactively managing long-term divergence. That implies either application state is being directly exposed to the user, or the application code is using an API to present version control affordances. We encourage both communities to think outside the box of Git, which has proven to baffle not only end-users but also a substantial fraction of developers.

8 Challenge goals

Text from the Extract Entity challenge that maybe should be lifted up to the top-level discussion of goals

The Extract Entity challenge for live programming is to provide interactive operations in the programming environment that will perform the schema change on live code and data. The data can be in any form desired, including relations, objects, or JSON. The code can be a simple CRUD UI that also provides the shipping department its view of unshipped orders. The central problem is to coordinate the schema change with code edits/refactorings to keep the system executing live and correctly. Solutions should try to minimize:

1. The number and complexity of commands or UI affordances that must be introduced.
2. Interruption to live execution of the code.

[2] If only we had this in the 90's when Smalltalk had a shot at the mainstream!

The Extract Entity challenge for local-first software is similar to the live programming challenge except that the context is an application that offers similar functionality with peer-to-peer collaboration. In this context the schema change does not need to be performed interactively – a developer can take some time to build, test, and package a solution, perhaps using an API or DSL. The hard part comes when the schema change is to be deployed across all the replicas. This deployment must synchronize code and data upgrades (or decouple them as in Cambria [**Cambria**]). The deployment must also deal with migrating "in flight" operations so that all replicas converge on the same state without data loss, and ideally without centralized coordination. A radical approach could try to incorporate schema change operations into the underlying datastore/CRDT itself and integrate code as well, but layered solutions are welcome too.

8.1 Goals

Solutions should try to minimize:

1. The need for users to manually intervene.
2. Possibilities that developer-written code is subtly incorrect in edge cases, preferably by reducing the need for developer-written code.
3. The need for a centralized coordinator.
4. How long replicas might be partitioned until resynchronization is acquired.

9 Related Work

Schema evolution has long been a major problem for SQL databases, which provide suprisingly little help. SQLite [**sqliteDatatypes**] uses dynamically typed values allowing them to be implicitly converted if the datatype of the column changes. MYSql [**mysqlAlterTable**] can reorder columns without destroying their data. But apart from such special cases, SQL databases have no general purpose support for data migration. As a result in practice schema evolution is still mostly done by writing custom SQL code to alter the schema and migrate the data. Such custom code is greatly complicated if it needs to be done without taking the database offline or must be coordinated across multiple shards. In *Refactoring Databases* **ambler06** [**ambler06**] offer a comprehensive taxonomy of schema evolution patterns, including typical strategies for online migration and sample SQL implementations.

bernstein07 [**bernstein07**] observed in 2007 "There are hardly any schema evolution tools today. This is rather surprising since there is a huge literature on schema evolution spanning more than two decades." There are now more tools available. Some tools such as Liquibase [**liquibase**] and PlanetScale[**planetscale**] could be characterized as version control and continuous integration/deployment for schema. They track schema changes and can calculate diffs in the form of SQL DDL statements to convert one schema version to another, but do not help migrate data. Unfortunately comparing schemas can be ambiguous about the intention of changes. For example

has a column been renamed or has it been deleted and a new one created? That distinction makes a big difference to the data in that column. EvolveDB[**evolvedb**] addresses this ambiguity by reverse-engineering the schema into a richer data model and tracking the edits to that model within an IDE. This more precise edit history can be used to infer higher level intentions of a schema change, which then generate SQL scripts to evolve the database. EdgeDB [**edgedb**] resolves ambiguities by asking questions of the developer, with some answers supplying custom migration code in a proprietary query language.

Some tools provide a Domain Specific Language (DSL) to describe schema evolution. Rails Migrations [**RailsMigrations**] embeds a DSL in Ruby to manage schema migration but is comparable to the capabilities of SQL DDL, often requiring the addition of custom Ruby or SQL code. **curino08** [**curino08**] spawned a stream of research on Database Evolution Languages (DEL) by defining a Schema Modification Operator (SMO) as "a function that receives as input a relational schema and the underlying database, and produces as output a (modified) version of the input schema and a migrated version of the database". SMOs can also rewrite queries to accomodate schema changes. **herrmann15** [**herrmann15**] defined a relationally complete DEL and then extended it into a Bidirectional Database Evolution Language (BiDEL) [**herrmann17**]. BiDEL appears capable of handling the basic requirements of our *Extract/Absorb Entity* and *Multiplicity Change* challenges, though *Split/Merge Entity* is less clear. It provides schema divergence by supporting multiple schema within one database, but would need some extension to handle data divergence.

Schema evolution is also a problem for NoSQL [**sadalage12**] databases. While such databases are sometimes called "schemaless" in effect that means the schema is left implicit and tools must try to infer it [**storl20**, **storl22**]. **Cambria** [**Cambria**] uses lenses [**Foster2007**] for bidirectional transformation of JSON. **scherzinger13** [**scherzinger13**] define a set of operators like the relational SMOs discussed above. **chillon21** [**chillon21**, **chillon22**] offer a more comprehensive set of SMOs that may be capable of handling *Extract/Absorb Entity* and *Multiplicity Change* but not *Split/Merge Entity* nor divergence. None of the above approaches to NoSQL evolution have yet extended into updating code or rewriting queries like their SQL cousins.

Schema evolution has also been studied for Object Oriented Databases(OODB) [**li99**, **banerjee87**]. Smalltalk [**Goldberg80**] is itself an OODB, persisting all object instances in an "image file" with some evolution capabilities incorporated in the programming environment [**Goldberg80**]. Gemstone turns the Smalltalk image into a production-quality database and accordingly provides a complex schema evolution API [**Gemstone**].

10 Extract/Absorb/Split/Merge Entity

Maybe called *Individuation*? Maybe called *Entity Evolution*?

The Acme Corporation needs to record orders from various customers for various products. The simplest implementation is a spreadsheet with a row for each order and columns for information about the customer and product. It might look like this:

order_id	item	quantity	ship_date	customer_name	customer_address
1	Anvil	1	2/3/23	Wile E Coyote	123 Desert Station
2	Dynamite	2		Daffy Duck	White Rock Lake
3	Bird Seed	1		Wile E Coyote	123 Desert Station

Table 1 Orders

The shipping department filters this table on blank ship dates to see what they need to ship. But after a while the orders department realizes they are wasting effort duplicating the address for new order from an old customer. And when the customer's address changes they have to go back and edit all of their orders. What is needed is two tables, one with orders that links to one with customers, like this:

customer_id	customer_name	customer_address
1	Wile E Coyote	123 Desert Station
2	Daffy Duck	White Rock Lake

Table 2 Customers

order_id	item	quantity	ship_date	customer_id
1	Anvil	1	2/3/23	1
2	Dynamite	2		2
3	Bird Seed	1		1

Table 3 Orders linking to Customers

We often encounter this situation when we realize that the attributes of one type of entity should actually belong to a distinct type of entity that will be associated with the first one. Often the motivation is to centralize changes to those attributes in one place. That is what it means to be an *entity*: a referenceable holder of mutable attributes. We call this operation *Extract Entity*.

Unique identifiers (here consecutive numbers) have been assigned to the customers and are referenced from the orders. This allows, for example, a spelling error in a customer address to be fixed in one place. It also allow a customer name or address to change without breaking the connection from all associated orders. Unique identifiers are the essence of entities, whether they are uniquely generated primary keys in a database or an abstraction of a memory address in a programming language.

The *Extract Entity* operation must 1) merge duplicates, 2) assign unique identifiers, and 3) reference these identifiers from the orders. Our example took the simple approach of merging entities whose attributes are all equal. But that might not be

correct in all cases. What if there was a typo in one of the addresses that made it look different from other instances of the customer? To repair that error we need to merge the falsely distinct customers into one and fix any references from orders. We call this operation *Merge Entities* – note that it is not a schema change but a data change, yet nevertheless not commonly supported in data APIs.

Acme Corp has a problem, for spreadsheets don't naturally support relationships between entities. End-user databases like Airtable [**airtable**] and Notion [**notion**] support *links* between tables but do not provide operations that can do an *Extract Entity*. Neither do the established database products (see Related Work). Acme will need to write code or do a big manual conversion.

It is natural to expect every schema evolution to have an inverse, for the change in requirements that triggered an evolution might change back again, and we ought to pay off the technical debt of obsolete design decisions. Research on *Divergence Control* [**Foster2007**, **herrmann17**, **chillon22**] further suggests operations should be invertible. The inverse of *Extract Entity* is *Absorb Entity*. It raises the question of how to handle *one-to-many* relationships. For example should absorbing orders into customers do a relational join, returning us back to the starting point, or should it nest orders within customers, as a NoSQL database might do?

The inverse of *Merge Entities* is *Split Entity*, but that can be awkward. Imagine that we didn't have addresses for customers to disambiguate them with and only knew their non-unique names. Then *Extract Entity* will conflate them and discard the information needed to properly invert the operation. Should this information be retained? One could object that we shouldn't have taken orders for people we can't uniquely identify yet. And even if that was necessary it means the customers really aren't distinct entities and shouldn't have been extracted. These are arguments that we shouldn't need *Split Entity*, contradicting the principle that evolution operations should be invertible. It is interesting to observe that evolution in nature can also fail to invert, leaving behind vestigial features. We consider this to be an unsettled issue.

10.1 Code Coevolution

A simple example of how code should evolve along with the schema is a SQL query to report all pending orders in the original schema:

```
SELECT order_id, item, quantity, customer_name, customer_address
FROM Orders WHERE ship_date IS NULL;
```

The *Extract Entity* operation described earlier should rewrite this query into:

```
SELECT order_id, item, quantity, customer_name, customer_address
FROM Orders
JOIN Customers ON Orders.customer_id = Customers.customer_id
WHERE ship_date IS NULL;
```

Alternatively the *Extract Entity* operation could create a view in which the original query is executed.

10.2 Goals

A solution to this challenge should:

1. Define the *Extract Entity* operation.
2. Define the *Absorb Entity* operation and motivate the tradeoff between joining and nesting.
3. Define a *Merge Entities* operation or if using automatic deduplication then discuss how to handle mistakes.
4. Define the *Split Entity* operation or justify why it is not needed.
5. If proposing a user-facing solution explain the UI affordances invoking these operations.
6. Extend these operations to the Divergence Control scenario.

PROGRAMMING CONFERENCE 2023
Invited speakers

- Adele Goldberg, adele@xerox.com
- Margaret Hamilton, hamilton@mit.com
- Betty Jean Jennings, betty@rand.com

(a) Initial state of the document.

PROGRAMMING CONFERENCE 2023
Invited speakers

- Ada Lovelace, ada@rsoc.ac.uk
- Adele Goldberg, adele@xerox.com
- Betty Jean Jennings, betty@rand.com
- Margaret Hamilton, hamilton@mit.com

(b) Added a speaker and sorted the list.

PROGRAMMING CONFERENCE 2023
Invited speakers

Name	Email	Who
Adele Goldberg	adele@xerox.com	TP
Margaret Hamilton	hamilton@mit.com	JE
Betty Jean Jennings	betty@rand.com	JE

(c) Refactored list into a table.

PROGRAMMING CONFERENCE 2023
Invited speakers

Name	Email	Who
Ada Lovelace	ada@rsoc.ac.uk	
Adele Goldberg	adele@xerox.com	TP
Betty Jean Jennings	betty@rand.com	JE
Margaret Hamilton	hamilton@mit.com	JE

(d) Resulting document after change merging.

Figure 2 Conference organizer. Initial version of the document with two independent changes and the final version of the document after the two changes are merged.

11 Conference Organizer: Merging edits in computational documents

In this section, we consider challenges related to working with a document-based format akin to HTML. We discuss two variants of the format. In the first format, a document contains only data. In the second variant, the format is extended with computed values. A computed value is a node in the document whose value is specified by a formula akin to those used in spreadsheets. The equations can refer to other nodes in the document, possibly using an absolute or a relative path and may involve a range of built-in functions, for example to count the number of elements or to sum a sequence of numerical values.

We assume that the document can be concurrently edited by multiple users in the same fashion as in local-first software, i.e., users may edit a local copy that then needs to be synchronized with changes made by other users. The edits to the document can be done in a rich-text editor, or by using a more structure-aware structure editor that can, for example, record changes as a sequence of high-level edit operations.

The concrete challenges are centered around the problem of collaborative conference planning as illustrated in Figure 2. A number of co-organizers are planning a conference that will feature talks by several invited speakers. They need to agree on the speaker list, contact speakers and calculate the conference budget.

Framework perspective. In this scenario, the only transient *state* is the result of computed values. The content of the document corresponds to the *data* pace layer. A change in the data layer may invalidate the computed value at the state layer. Formulas correspond to the *code* layer and changing them, again, invalidates the state. The *schema* pace layer is implicit. A change to the document that modifies content is a mere data change, but restructuring the document is an implicit schema change. Such schema change that may affect the code pace layer.

The scenario assumes the *decentralized* model. There are multiple program (document) variants and it should be possible to propagate a change at any pace layer to other program variants. The challenge may be solved using both the easier *convergence* and the more challenging *divergence* model.

Applications. One of the earliest systems to combine documents with computational elements is Boxer [**diSessa86**]. The challenges in this section apply to a wide range of systems that can be seen as its successors. The specific scenario discussed here is inspired by the use of Notion [**notion**], which is a collaborative structured document editing system. Treating documents as end-user programs has recently been explored in Potluck [**Litt2023**], which is local-only but includes formulas similar to those in our challenge.

The challenges in this section are also relevant to work on computational scientific notebooks. Jupyter [**Kluyver2016**] uses a simple representation as a sequence of cells and does not have a mechanism for embedding data in the document and referring to it, but numerous project such as Nextjournal [**Nextjournal21**] advance the format. The idea of producing interactive, computational secientific documents is an active research area. Recent contributions include Nota [**Crichton2021**] and Living Papers [**Heer2023**].

Challenge #1: Merging structured document edits

First, the organizers need to agree on a list of speakers to invite, coordinate who contacts whom and track the acceptance of invitations. They start with a document shown in Figure 2a. The challenge is to merge document edits that are done locally and independently by two co-organizers. Specifically, consider the following two edits:

1. The first organizer adds an additional speaker to the list and sorts the list of speakers alphabetically by their first name. (Figure 2b)
2. The second organizer refactors the list into a table. They split the single textual value into a name and an email (using a comma as the separator) and add an additional column for tracking which of the organizers should contact the speaker.

The system should be able to merge the changes and produce a final table shown in Figure 2d. The resulting table needs to include the additional speaker created by the first organizer, use the order specified by the first organizer and use the format defined by the second organizer. The newly added speaker should be reformatted

into the new format. For the newly added "Organizer" column, the second user may specify default value or the newly added row may use an empty value.

Framework perspective. In this challenge, the change done by the first organizer is at the *data* layer, while the change done by the second organizer spans the *schema* and *data* layers. In implementing the schema change, the second organizer transforms the affected document data. The challenge is to apply the same transformation to the data independently added by the second organizer.

Requirements. The challenge is concerned with merging two sequences edits to a base document. A solution to the challenge should strive to satisfy the following goals:

- *Completeness.* It should be possible to semi-automatically merge any two sequences of edits. In other words, the merging should always be defined, regardless of what edits the users perform, although user input may sometimes be necessary.
- *Commutativity.* The order of edits should not matter, regardless of their pace layer. If users *A* and *B* perform edits independently starting from the same initial document, the resulting document should be the same if the system treats edits from *A* as occurring before the edits of *B* and vice versa.
- *Conflict resolution.* It may not always be possible to merge conflicting edits. In this case, the system should interactively ask the user for guidance or, possibly, use default behavior specified by the user.
- *Structure editing.* The system may use a structure editor that provides high-level commands for operations such as reordering of elements or refactoring of a list into a table. In other words, we recognise that the challenge may not be solvable in a system based on plain text editing of document source code.

Remarks. As discussed earlier, this challenge does not make an explicit distinction between the schema of the document and data. The implementing system may or may not maintain such distinction. In our example, adding a new speaker and sorting the list is a mere change of the data, but the refactoring of a list into a table is a schema change. The system may offer different user interface for changes at different layers and leverage knowledge about such distinction, for example, by handling the merging of schema-related and data-related changes differently.

Challenge #2: Applying schema edits in presence of code

This challenge extends the previous problem. The conference organizers now also want to use the document to manage the conference budget. In a very simplified form, one aspect of this is calculating the estimated travel expenses for the speakers. For this, they add a computed value to the document that counts the number of speakers and multiplies the result by a fixed cost per speaker elsewhere in the document.

The scenario is shown in Figure 3a. Assume two co-organizers start with the same original document shown in Figure 2a. One adds an additional section with computed

PROGRAMMING CONFERENCE 2023
Invited speakers

- Adele Goldberg, adele@xerox.com
- Margaret Hamilton, hamilton@mit.com
- Betty Jean Jennings, betty@rand.com

Conference budget
Travel cost per speaker:
`$1200`
Number of speakers:
`=COUNT(/ul[id='speakers']/li)`
Travel expenses:
`=/dl/dd[0] * /dl/dd[1]`

(a) Initial state of the document with formulas.

PROGRAMMING CONFERENCE 2023
Invited speakers

Name	Email	Who
Ada Lovelace	ada@rsoc.ac.uk	
Adele Goldberg	adele@xerox.com	TP
Betty Jean Jennings	betty@rand.com	JE
Margaret Hamilton	hamilton@mit.com	JE

Conference budget
Travel cost per speaker:
`$1200`
Number of speakers:
`=COUNT(/table[id='speakers']/tbody/tr)`
Travel expenses:
`=/dl/dd[0] * /dl/dd[1]`

(b) Resulting document after schema change merging.

Figure 3 Conference organizer. Formulas to calculate the budget have been added to the original document (Figure 2a) and those have to be merged with the refactoring of a list into a table (Fig 2c). Formulas in the final version are updated to match with the new document structure.

values, while the other performs the edits discussed earlier. The challenge is, again, to merge the edits done independently by the co-organizers.

1. The first co-organizer adds the budget calculation as shown in Figure 3a. This includes two formulas. The first selects all `li` elements of a `ul` element with ID `speakers` (not visible in the UI) and counts their number. The second formula multiplies the constant travel cost per speaker by the number of speakers. Here, we assume the new section uses the HTML definition list `dl` and the formula selects its first and second `dd` item, respectively.
2. The second co-organizer performs the refactoring and data edits discussed previously. They edit the speakers and refactor the structure to use a table, resulting in the document shown in Figure 2d.

As before, the system should be able to merge the edits made to the document.

Framework perspective. In this challenge, the change done by the first organizer is at the *code* layer, while the change done by the second organizer is both at the *data* layer and, implicitly, at the *schema* layer. A change solely at the *data* layer would not affect code, but a change at the *schema* layer requires adapting some aspects of the code.

Remarks. Addressing the challenge requires that the refactoring of the document structure done in (1), is also reflected in the code of the equations added in (2). As shown in Figure 3b, in this specific case, the system needs to change the equation

COUNT(/ul[id='speakers']/li) to COUNT(/table[id='speakers']/tbody/tr). This is the case because the refactoring consists of four steps that each transform the document and also need to update the code of the equation accordingly:

- Change the type of the ul element to table
- Wrap the body of the table element with tbody
- Change the type of all children of tbody from li to tr
- Further split the body into two td elements

It is expected that the system solving the challenge will need to be aware of those high-level edits. In particular, they may be explicitly specified by the user (through an interactive user interface for editing the document) or semi-automatically inferred.

Requirements. A solution to the challenge should meet the goals of the preceding challenge, i.e., completeness (merging should be always defined), commutativity (order does not matter), conflict resolution (ask for guidance if needed), convergence (everyone eventually wants the same document) and structure editing (the system may capture high-level edits).

Challenge Extensions: Liveness and divergence

So far, we assumed that the system follows the *convergence* model, i.e., all variants of the program eventually adopt all changes. A more difficult variant on the challenge is to also support the *divergence* model. In this case, some of the users continue using their variant of the program without adopting all of the changes done by other users.

> **Framework perspective.** In the *divergence* model, it may be expected that users will want to adopt all changes done at the *data* pace layer. They may selectively choose some of the changes at the *code* and *schema* layers. The challenge is maintaining the same data across multiple program variants and merging changes done to the data based on different document schema.

The basic challenge outlined above focuses primarily on the local-first software scenario, but it can be extended to also apply to the live programming scenario. In particular, when the document contains computed values, those may be evaluated either explicitly by the user or automatically on-the-fly. An edit can then invalidate some of the values (either by changing the data that a computed value depends on, or by changing the equation used to compute the value). A live programming system should be able to detect which computed values are affected by an edit and, either invalidate those (requiring an explicit re-evaluation by the user) or automatically re-evaluate them (and possibly highlight the affected values to inform the user).

> **Framework perspective.** In the live extension, we also consider the transient *state* pace layer. A system should automatically detect when a change at the *data* or *code* layers affects the computed value and forces it to be recomputed. Interestingly, a

change at the *schema* layer may not always affect the *state* layer, even if it schema change requires modifying formulas that compute the state.

12 Live Modeling Languages Require Run-Time State Migration

12.1 Context

In stateful live programming, a change in the program can make the run-time state of the execution out of date, hence we want to migrate the run-time state to keep on running.

One can see a program itself as an instance of a static schema (its AST type), that in turn determines (defines/implies/induces) a run-time schema: the run-time structures of code (e.g., classes, inheritance links, declarations, method definitions etc.), as well as the structure of the run-time state. Whereas the run-time *code* structures do not typically change while using an application, the run-time *state* constantly changes. Live programming, however, requires reconciling changes to the program with the run-time structures of both the code and the state. This typically means that at a certain point (a quiescent point) during execution, the code structures need to be updated (hot swapped), and the state needs to be *migrated*.

12.2 Example

Consider the example of a simple state machine language with on-entry actions, and typed global variables (loosely inspired by the SML language of **vanRozen19** [**vanRozen19**]). An example state machine and an excerpt of its abstract syntax schema is shown in Figure 4. Depicted on the left, an actual statemachine modeling the opening and closing of a door, with one global boolean variable, `isClosed`, which is flipped according to transitioning between the `closed` and `opened` states. For the sake of brevity, the abstract syntax schema (similar, to e.g., and Ecore metamodel [**EMF**]) shown on the right of the figure omits classes for the on-entry actions (`Stmt`) and variable types (`Type`).

In the following we assume that the state machine is executed by an interpreter that process an incoming stream of events, and traversing run-time pointers representing transitions accordingly, executing on-entry actions, where the notion of the current state and the values of the global variables are the collective run-time state of the program. The structure that the interpreter operates on can be described as derived schema from the abstract syntax scheme shown on the right of Figure 4, but specialized for a particular machine, e.g., `Doors`. This schema will have additional fields and associations modeling the run-time data required for execution. For the `Doors` statemachine, this means adding the current state to the `Machine` type (required for any machine) and the machine specific data fields, namely `isClosed`. The run-time schema for `Doors` is then the following[3]:

```
1 class Machine++
```

[3] It is possible to represent the global variables at run-time using an untyped table or hashmap, but for the sake of schema migration complexity, the encoding as fields in classes is more instructive.

```
machine Doors

var isClosed: bool = true

state closed
  isClosed := true
  on open => opened

state opened
  isClosed := false
  on close => closed
```

```
class Machine
  name: string
  states: State*
  vars: Decl*

class State
  name: string
  onEntry: Stmt*
  transitions: Trans*

class Trans
  event: string
  target: State

class Decl
  name: string
  type: Type
```

Figure 4 A statemachine and its abstract syntax schema (omitting Stmt and Type)

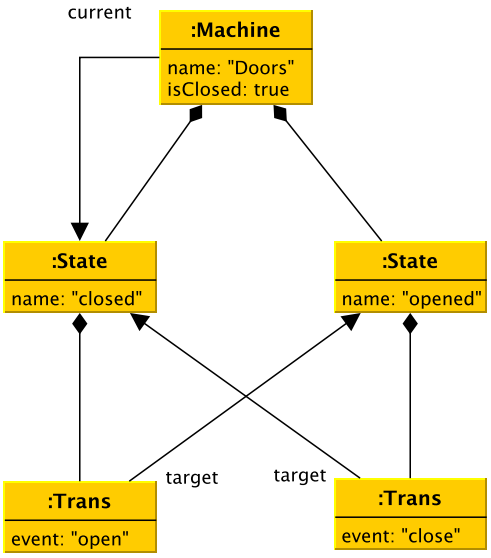

Figure 5 UML object diagram of the run-time structures of the Doors statemachine (excluding code objects)

```
  states: State*
  current: State
  isClosed: bool
```

This *extended* class Machine^{++} also represents state machines, but this time at run time.

An instance of a running state machine consists of an object graph conforming to the run-time schema of state machines, updating the current state pointer in response to events. This object graph is graphically depicted in Figure 5 as a UML object diagram,

Change	What	How
add	state	simple
remove	state	structurally simple, but heuristic if state is current
rename	state	simple
add	variable	migrate class and initialize field
remove	variable	migrate class (NB: assumes variable is unused)
rename	variable	rename in class, preserving value
type change	variable	migrate class and convert or reinitialize value
add	transition	simple
remove	transition	structurally simple, but heuristic for pending events
event change	transition	structurally simple, but heuristic for pending events
add/remove	statement	hotswap code at quiescent point of interpreter

Table 4 Possible program changes and how to deal with them at run time

excluding the statement objects representing on-entry actions. As the diagram shows, the running machine has a current state (`closed`) and the `isClosed` field has an actual value (`true`). It is instructive to note that the diagram encode both static and dynamic aspects of the state machine language. Live editing the state machine then requires to *patch* [**SemanticDeltas**] this run-time structure without shutting it down, possibly requiring migration of run-time state.

12.3 Challenge

The space of possible (well-formed[4]) changes to a state machine are summarized in Table 4. The first column indicates the change category, the second the affected kind of object, and third a short description of how to deal with the respective change category. Some notes about this table:

- all rows with "simple" in the third column only require a quiescent [**Tranquility**] point in the interpreter loop to update the run-time structure (e.g., as shown in Figure 5).
- removing a state, however, is only *structurally* simple, since removal is easy, but special care is required if the subject of removal is the current state. In this case, some heuristic is needed, such as: reject the edit, point the current state to the initial state, or some other strategy (e.g., the nearest state, previous state etc.)
- everywhere a row mentions "migrate class" in the third column, data migration is needed: the run-time objects conforming to the old class must be transformed to instances of the updated class, similar to how Smalltalk migrates objects using `become:`.

[4] The abstract syntax schema is not expressive enough to define all static invariants of the language, such as: states must by unique by name, references variables in actions must be declared, etc.

- removing a transition or changing the event of a transition potentially has to deal with pending events (e.g., in an event queue) expecting such transitions, since such events are now potentially stale. Strategies to deal with this situation include: simply dropping the events, or requiring that such edits cannot be patched at run time when there are pending events.
- the type change edit of variable requires a strategy for the current value: either discard and reinitialize, or perform value conversion. For instance, if the type change is from boolean to integer, then true could become 1, and false could become 0.
- note finally that rename variable is mentioned explicitly as a change: this makes it possible to preserve the run-time value of the variable.

12.4 Goals

The goals for this challenge are twofold: from the end-user perspective and from the language engineering perspective. A particular challenge from the end-user perspective is to "do minimal harm": it is essential for fluid programming experience that the automatically triggered migrations are in a sense as near as possible to the previous application state, to not surprise or confuse the user/developer. One approach considered in earlier work [**RuntimeConstraint**] employs the constraint solver Z3 to find a "nearest" run-time instance compatible with a source change. Nevertheless, the patching of the run-time state should be quick enough so as to not disrupt the programmer experience.

From the language engineering perspective the goal is to employ techniques, formalisms, and tools, to make the construction of such languages easier. The above example state machine DSL is derived from earlier work [**vanRozen19**], where the authors manually implemented the run-time patch operation and concluded that even for such a simple language (simpler even than the example above) it is a complex and error-pone endeavor. Furthermore, the field of software language engineering studies and develops generic and reusable techniques to improve the development of DSLs and programming languages, for instance, in the context of language workbenches [**ERDWEG201524**]. The development of live programming languages, however, is currently still out of reach for all existing language workbenches. The aforementioned approach using a constraint solver is an example of such a *language parametric* technique, in that it operates on the (extended) abstract syntax metamodel of a language, and does not assume anything further about the language itself. Another approach is the Cascade metamodeling formalism which has builtin support for run-time patching [**Cascade**]. However, further research is needed to design better principled language engineering approaches that solve the problem in a way that is both declarative and fast.

12.5 Extensions

While the above example is arguably simple, the problem becomes much more challenging when the programs themselves define data types, classes, records, structures

etc. Since possibly many instances (values, objects) of such data types may exist at run-time, these all have to be migrated in such a way that programmer experience is minimally disrupted, and that the invariants of said data types is maintained. Another extension, tying in with the data-oriented examples above, involves refactorings of data types in a program. Typically a refactoring should be behavior preserving, but can it also preserve run-time data? The minimal example is the consistent variable rename rename in Table 4, which should not have any effect on run-time state. A more complex refactoring is described in Section 13.

13 Live programming in the context of Elm architecture

13.1 Context

Another challenge involving stateful live programming can be drawn from the programming model based on the Elm architecture. In this model, a reactive (web) application is structured in terms of current state and events that affect the state. The implementation then consists of two functions that we may call `update` and `render`:

```
type State = { .. }
type Event = .. | ..

val update : State -> Event -> State
val render : State -> Html
```

The programming model works as follows:

- `State` represents the entire application state, i.e., everything that the user can work with.
- `Event` represents all events that the user can trigger by interacting with the application.
- `update` is called whenever an event happens. It takes the current state, the event and computes a new state.
- `render` takes the current state and produces a representation of what should be displayed on the screen (e.g., an HTML tree).

Traditionally, when Elm applications are developed, they are restarted each time the code is modified and any previous state is discarded. However, a more effective programming model based on live programming would allow live updates to both the code of the two functions and the structure of the two types.

13.2 Example

As an example, consider the uninspiring, but well-known, TODO list application. The types that capture the state and events in the application may look as follows:

```
type Item = { id : id; title : string; completed : bool }
type State = { items : Item list }
```

```
type Event =
  | SetCompleted of id * bool
  | SetTitle of id * string
  | Remove of id
  | Add of string
```

The application state consists of a list of items. Each item has a unique ID alongside with a title and a flag indicating whether it is completed. The events represent edits to the items, deletion and addition. The implementation of the update and render functions is simple and not important for the challenge.

13.3 Challenge

Now, imagine that we have a running TODO list application with the above state and events. To test the application, the programmer has already created a number of items and so there is a single value of the State type that represents a current state of the application such as:

```
{ items : [
    [ { id = 1; title = "check twitter"; completed = true }
      { id = 1; title = "Write paper"; completed = false } ] }
```

As above, there are a number of edits to the code and types that the programmer may want to do without restarting the application. Those are summarized in Table 5. Modifying the code is simple and only requires waiting until the current execution completes. Modifying the Event type is also simple, but it may lead to unused code or missing case in update that needs to be addressed. Finally, modifying State ranges from relatively simple problems (adding a new field) to challenging case when the structure of State is changed.

For example, imagine that the programmer would want to edit the structure of the state and migrate the previous definition of State to the following new definition where individual fields are stored in separate lists:

```
type State =
  { ids : id list
    titles string list
    completes : bool list }
```

This representation is semantically equivalent (assuming the lists have the same length) to the original one. It should thus, in principle, be possible to migrate the original state value to a value using the new structure. Moreover, it should, in principle, be also possible to automatically transform the implementations of update and render to work as before, but using the new state structure.

13.4 Goals

As above, the key requirement from the live programming perspective is to “do minimal harm”. In this case, this results in the following goals:

Change	What	How
modify	render	simple, but wait until current execution finishes
modify	update	simple, but wait until current execution finishes
add	case to Event	requires adding corresponding case to update
remove	case from Event	remove unused code from update
add	field to State	migrate state value and initialize field
remove	field from State	migrate state (assuming field unused)
modify	structure of State	migrate state value and edit code accordingly

Table 5 Possible program changes and how to deal with them at run time

- When migrating the application state, this needs to be done automatically and the system should strive to produce new state that is as near as possible to the previous state.
- The solution may rely on structure editing so that the system has access to a high-level logical description of the edits performed by the user.
- The system needs to handle the case when the type structure diverges from the code structure. This can be addressed in various ways (transform code, add error handlers, etc.) but it is desirable to avoid "breaking" the render function as this would make the new application state impossible to see.

13.5 Remarks

It is worth noting that the particularly difficult aspect of this challenge, i.e., the case where the structure of state is refactored to an equivalent one, is related to the Extract Entity challenge.

14 Multiplicity change

14.1 Context

A common kind of schema change is *multiplicity change*: when a field changes from storing a single value to storing multiple values. For example, we might start by storing a single address for each contact in an address book, before realizing that we need to store multiple addresses for a single contact; or we might assign each todo in a list to one person before realizing we want the ability to assign a todo to multiple people.

In a relational schema, solving this problem might require extracting a normalized table for the linked entity; some of the challenges of this approach are covered in the Extract Entity challenge. In this section we will instead consider a document schema with support for arrays; in this context, we can have a multiplicity change by turning a scalar value into an array value.

Multiplicity change clearly requires changing both the data and the code. It becomes particularly challenging to handle when there is ongoing divergence between multiple versions of the data.

14.2 Example

Consider the following schema for a todo list item, in which each item has a single assignee, represented by a user ID:

```
type Item = { id : id; title : string; assignee : string }
```

Now, we change the schema so that each item has a list of assignees:

```
type Item = { id : id; title : string; assignees : Array<string> }
```

Our goal is to preserve the ability for actors in the system to read and write to a shared todo list in either the old or new schema – for example, they should be able to write to either the scalar assignee field or to the list of assignees.

A natural invariant to preserve across these schemas is that the value of the scalar field should equal the first element of the list field (and if the list is empty, the scalar field should be null). If we were to write code for a one-time data migration from the scalar to list schema, we could easily satisfy this invariant:

```
item.assignees = if (item.assignee == null) then [] else [item.assignee]
```

Ongoing edits to the array schema can also be handled in a straightforward way. After edits are made, the value of the scalar field should be set to the first element of the new array (or null if the new array is empty.)

However, handling edits to the scalar schema presents more of a challenge. Consider the following todo with two assignees, presented in terms of the scalar and array schemas. A write is made in the scalar schema to set the new assignee to C.

```
todoScalar = { id: 1, title: "Foo", assignee: A }
todoArray = { id: 1, title: "Foo", assignees: [A, B] }

todoScalar.assignee = C
```

What should the array value become after this edit to the scalar schema? To satisfy our invariant, we know that C must become the first element of the array, but this leaves open several options with different tradeoffs:

1. [C]: "Only C should be assigned." This option produces an array that corresponds directly to the resulting scalar. But it has the downside of deleting data that wasn't even visible in the scalar schema.
2. [C, B]: "Replace A with C." If the writer wanted to remove A from the assignment and add C, this option performs that intent. However, there is data remaining in the list which was not visible to the scalar writer.

3. [C, A, B]: "Add C to the list." Perhaps the scalar writer wanted to add C to the assignment without removing anyone; this option satisfies that intent. But it preserves data not visible to the scalar schema.

Because the intent of the writer to the scalar schema cannot be unambiguously interpreted from the write alone, it is impossible to make a perfect choice among these options.

14.3 Remarks

If schemas, code and data can all be updated atomically together, then multiplicity change is relatively straightforward to handle. Existing scalar data can be trivially migrated to a list, and the code using the data will need to change to accommodate the list data.

However, ongoing divergence makes multiplicity change much more difficult, and exposes some general challenges. Different schemas may expose partial information, and writers in those schemas have to operate without total knowledge of the information available in other schemas. As a result, synchronizing writes across the schemas requires making difficult tradeoffs, such as choosing to either preserve or destroy hidden data not visible to the writer.

Although there is likely no silver bullet to navigate these tradeoffs, a good solution to this problem would give developers or users tools to manage these tradeoffs. For example, a system might allow developers to specify the desired behavior for a particular pair of schemas based on the requirements of the domain. Or a system might even allow users to manually disambiguate their intent for a given write.

About the authors

Jonathan Edwards is TBD. Contact him at jonathanmedwards@gmail.com.

Tomas Petricek is TBD. Contact him at tomas@tomasp.net.

Tijs van der Storm is TBD. Contact him at storm@cwi.nl.

Geoffrey Litt is TBD. Contact him at gklitt@gmail.com.