Data Migration Under a Changing Schema in Ampersand

Sebastiaan Joosten and Stef Joosten

August 29, 2024

Data Migration under a Changing Schema in Ampersand

Sebastiaan Joosten[1][0000−0002−6590−6220]
and Stef Joosten[2,3][0000−0001−8308−0189]

[1] University of Minnesota, Minneapolis, USA
[2] Open Universiteit, Heerlen, the Netherlands
`stef.joosten@ou.nl`
[3] Ordina NV, Nieuwegein, the Netherlands

Abstract. Software generators that compile and deploy a specification into a functional information system can help to increase the frequency of releases in the software process. They achieve this by reducing development time and minimizing human-induced errors. However, many software generators lack support for data migration. This can inhibit a steady pace of releases, especially for increments that alter the system's schema in production. Consequently, schema-changing data migrations often face challenges, leading developers to resort to manual migration or employ workarounds.
To address this issue, this paper proposes a foundational approach for data migration, aiming to generate migration scripts for automating the migration process. The overarching challenge is preserving the business semantics of data amidst schema changes. Specifically, this paper tackles the task of generating a migration script based on the schemas of both the existing and the desired system, under the condition of zero downtime. The proposed solution was validated by a prototype demonstrating its efficacy. Notably, the approach is technology-independent, articulating systems in terms of invariants, thereby ensuring applicability across various scenarios. The migration script generator will be implemented in a software generator named Ampersand.

Keywords: Generative software · Incremental software deployment · Data migration · Relation algebra · Ampersand · Schema change · Invariants · Zero downtime

1 Introduction

In practice, information systems[4] may live for many years. After they are built, they need to be updated regularly to keep up with changing requirements in a dynamically evolving environment. Schema changes cannot always be avoided when updating software, so a *schema-changing data migration* (SCDM) will be

[4] In the sequel, the word "system" refers to the phrase "information system", to simplify the language a little.

necessary from time to time. For example, adding or removing a column to a table in a relational database adds to the complexity of migrating data. Even worse, if a system invariant changes, some of the existing data in the system may violate the new invariant. In practice, data migrations typically follow Extract-Transform-Load (ETL) patterns [16], for which many tools are available. However, ETL tools typically provide little support for invariants that change, forcing development teams to write code. In a world where automation of the software process is resulting in higher productivity, more frequent releases, and better quality, SCDMs should not stay behind. Roughly half of the DevOps [2] teams that responded in a worldwide survey in 2023 [7] are deploying software more frequently than once per day. Obviously, these deployments are mostly updates of existing systems. The risk and effort of SCDMs explains why these teams try to avoid schema changes in the first place. Our research aims at automating SCDMs to make them less risky and less costly, so development teams can deploy schema changes with zero downtime.

Data migration for other purposes than schema change has been described in the literature. For instance, if a data migration is done for switching to another platform or to different technology, e.g. [3,19], migration engineers can and will avoid schema changes and functionality changes to avoid introducing new errors in an otherwise error-prone migration process. In another example, Ataei, Khan, and Walkingshaw [1,17] define a migration as a variation between two data structures. They show how to unify databases with slight variations by preserving all variations in one comprehensive database. This does not cater for schema changes, however. Then there are SCDMs in situations without a schema or an implicit schema, e.g. [6]. Such situations lack the error preventing power that explicit schemas bring during the development of software. All errors a schema can prevent at compile time must then be compensated by runtime checks, which increases the likelyhood of end-users getting error messages. This requires versioned storage of production data and an overhead in performance. That is why this paper focuses on SCDMs for systems with an explicit schema. The prototypical use case for that is to release updates of information systems in production, where the semantic integrity of data must be preserved across schema changes. Another use case is application integration for multiple dispersed data sources with explicit schemas.

A practical complication in many data migration projects is the presence of deteriorated data. To clean it up may incur much work. Some of that work must be done before the migration; some can wait till after the migration. In all cases, data pollution in an existing system requires careful analysis and planning. We can capture the automatable part of the data quality problem by regarding it as a requirement to satisfy semantic constraints. E.g. the constraint that the combination of street name, house number, postal code, and city occurs in a registration of valid addresses can be checked automatically. In a formalism like Ampersand [10,11], which allows us to express such constraints, we can add constraints for data quality to the schema. This allows us to signal the data pollution at runtime. Some forms of data pollution will need to remain out of scope. An example is when a person has specified a false name without violating any constraint in the system.

The complexity of data migration has prompted us to develop an approach first, which we present in this contribution. We have validated the approach by prototyping because a formal proof of correctness is currently beyond our reach. This approach perceives an information system as a data set with constraints, so we can represent invariants (and thus the business semantics) directly as constraints.

The next section analyzes SCDMs with an eye on zero downtime and data quality. It sketches the outline of a procedure for SCDMs. Section 3 formalizes the concepts that we need to define the procedure. Section 4 defines the algoritm for generating a migration system, to automate SCDMs. Section 5 demonstrates the prototype of a migration system, which we used to validate our approach experimentally. For this purpose we have used the language Ampersand because its syntax and semantics correspond directly to the definitions in section 3.

2 Analysis

This section analyzes information systems qualitatively, to prepare for a formal treatment in section 3. The current section yields a procedure for migrating data from one system to another.

2.1 Information Systems

The purpose of an information system is to store and disclose data in a way that is meaningful to its users. Multiple users, working from different locations and at different moments, constitute what we will loosely call “the business”. The data in the system constitutes the collective memory of the business, which relies on the semantics of the data to draw the right conclusions and carry out their tasks.

Figure 1 depicts the situation before migration. The state of the system is represented by a data set, typically represented in some form of persistent store such as a database. An existing application service ingests traffic through an ingress and persists data in a data set. Our research assumes that the structure and business semantics are represented in a schema, from which the system is generated. Actors (both users and computers) are changing the data in a system continually. Events that the system detects may cause the state to change. To keep our approach technology independent, we assume that data sets contain triples. This makes our approach valid for every kind of database that triples can represent, including SQL databases, object-oriented databases, graph databases, triple stores, and other no-SQL databases.

We assume that constraints implement the business semantics of the data. Constraints represent business concerns formally, so they can be checked automatically and can be used to generate software. Some of the constraints require human intervention, while others require a system to intervene. In this paper, we distinguish three different kinds of constraints:

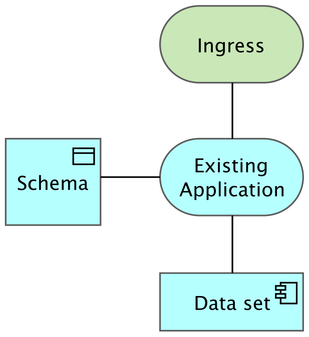


Fig. 1. Anatomy of an information system

1. Blocking invariant
   A *blocking invariant* is a constraint that is always satisfied in a system. It serves to constrain the data set at runtime. When the data set changes in a way that violates a blocking invariant, the system produces an error message and refuses the change.
2. Transactional invariant
   A constraint that can is kept satisfied automatically by taking corrective actions is called a *transactional invariant*. The system keeps these satisfied by adding or deleting triples to the dataset, typically using wrapped inside a classical database transaction to avoid issues with concurrency. As soon as the data violates a transactional constraint, the system restores it without human intervention. So, the outside world experiences this as a constraint that is always satisfied, i.e. an invariant.
3. Business constraint
   A *business constraint* is a constraint that users can violate temporarily until someone restores it. Example: "An authorized manager has to sign every purchase order." Every violation requires some form of human action to satisfy the business constraint (e.g. "sign the purchase order"). That takes some time, during which the constraint is violated. So, we do not consider business constraints to be invariants.

Summarizing, in our notion of information systems, concepts, relations, and constraints carry the business semantics. Of the three types of constraint, only two are invariants.

2.2 Ampersand

We employ Ampersand as a prototyping language to demonstrate our approach. Ampersand serves as a language for specifying information systems through a framework of concepts, relations, and constraints. It comprises the three types of constraints discussed in this paper, making it an ideal platform for practical testing of our approach. In Ampersand, developers articulate constraints using heterogeneous relation algebra [5,9]. The systems they generate keep invariants satisfied and alert users to violations of business constraints. The absence of imperative code in Ampersand scripts enhances reasoning about the system, while its static typing [18] yields the established benefits in software engineering processes [4,14]. Constraints carry business semantics, which makes "preserving the meaning as much as possible" explicit. An Ampersand script provides just enough information to generate a complete system, allowing extraction of a classical database schema (i.e., data structure plus constraints) from the Ampersand script.

Ampersand has seen practical use both in education (Open University of the Netherlands) and industry (Ordina and TNO-ICT). For instance, Ordina developed a proof-of-concept of the INDIGO-system in 2007, leveraging Ampersand for accurate results under tight deadlines. Today, INDIGO serves as the core information system for the Dutch immigration authority (IND). More recently, Ampersand played a role in designing the DTV information system for the Dutch Food Authority (NVWA), with a prototype built in Ampersand serving as a model for the actual system. TNO-ICT, a prominent Dutch industrial research laboratory, has utilized Ampersand for various research purposes, including a study of international standardization efforts of RBAC (Role-Based Access Control) in 2003, and a study of IT architecture (IEEE 1471-2000)[8] in 2004. Ampersand has also been employed at the Open University of the Netherlands, where it is taught in a course called Rule-Based Design[12]. Students in this course utilize a platform named RAP, constructed in Ampersand [13], which represents the first Ampersand application to run in production.

2.3 Zero downtime

To make the case for zero downtime, consider this problem: Suppose we have an invariant, u, in the *desired system*, which is not part of the *existing system.* In the sequel, let us call this a *new invariant.* Now, suppose the data in the existing system does not satisfy u. If u is a transactional invariant, the desired system will restore it automatically. But if it is a blocking invariant, the desired system cannot spin up because all of its invariants must be satisfied. To avoid downtime, we must implement new blocking invariants initially as a business constraint, to let users satisfy them. The moment the last violation of u is fixed, the business constraint can be removed and u can be implemented as a blocking invariant. This is the core idea of our approach.

The *migration system* to be generated is an intermediate system, which contains all concepts, relations, and constraints of both the existing and the desired system. However, it implements the blocking invariants of the desired system as business constraints. This migration system must also ensure that every violation that is fixed is blocked from recurring. In this way, the business constraint gradually turns into a blocking invariant, satisfying the specification of the desired system. Since the number of violations is finite, the business can resolve these violations in finite time. In this way, the migration system bridges the gap and users get a zero downtime SCDM.

Summarizing, the following requirements apply to SCDMs

1. users must experience zero downtime, to enable more frequent SCDMs.
2. users must be able to restore invariants in the new system. So, we need an intermediate "migration system" that implements blocking constraints of the desired system as business constraints, in order to deliver zero downtime.
3. the number of violations that users must fix is finite and decreases (monotonically) over time, to ensure that the migration system does not need to be kept alive infinitely.

2.4 Data Migrations

Data migration occurs when a desired system replaces an existing one, while preserving the meaning of the present data as much as possible [15]. In practice, data migrations typically deploy the existing system and the desired system side-by-side, while transferring data in a controlled fashion, as shown in Figure 2. To

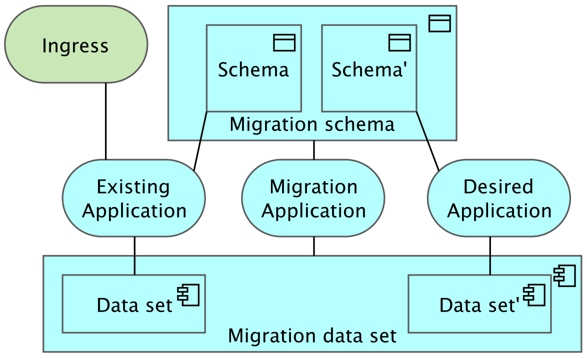


Fig. 2. Migration phase

automate the migration as much as possible and to achieve zero downtime, we must deploy a third system: the migration system. This system has its own schema. It comprises the schemas of both the existing system and the desired system, so the migration system can copy the data from the existing to the desired system. It uses transactions to copy the data and to resolve some forms of data pollution. Not all of the work, however, can be automated. Data pollution, new business rules, or known issues in the existing system may occasionally require tailoring a script describing the migration system to specific needs that require action of users in production. For that purpose, the migration engineer specifies business constraints in the migration system. In Ampersand, a developer can use business constraints to implement such needs.

Before a migration starts we assume that the existing system is up and running and all of its invariants are satisfied. The ingress is directing traffic to the existing system. The data migration has three distinct steps:

Pre-deploy The migration engineer deploys both the migration system and the desired system with their initial data, to allow both systems to satisfy their initial invariants before going live. The existing data is still in the existing system and the ingress still directs all traffic to the existing system. So, users notice no difference yet. The migration system starts copying data from the existing system.

Moment of transition (MoT) After the migration system is done copying data, the migration engineer switches all traffic to the migration system. This is the moment users will notice the difference because the traffic switch also deploys the functionality of the desired system. So, in the eyes of an average user, the migration system may look like the desired system. However, the migration system relaxes the blocking invariants of the desired system until users resolve the violations of the new blocking invariants. Since the existing system receives no more traffic, its activity will come to a halt and its data will become static. The migration system stays active until all invariants of the desired system are satisfied and the desired system can take over the work from the migration system.

Moment of completion (MoC) Once the invariants of the desired system are satisfied, the migration engineer switches all traffic to the desired system. The blocking invariants of the desired system are now in effect, so users cannot violate them anymore. After this switch, the migration engineer can safely remove both the migration system and the existing system.

Transactions in the existing system that happen during the time that the migration system is copying data cause no problem, because their changes are copied by the migration system, too. However, after the MoT there must be no new changes in the existing system to avoid violations of new invariants that the migration system has already fixed.

The following section introduces the definitions required to migrate data from one system to another.

3 Definitions

An *information system* is a combination of data set, schema, and functionality. For the purpose of this paper, we ignore functionality captured in user interfaces to focus on the transition of business semantics in the data. Section 3.1 describes how we define data sets. Since data sets are sets, we will use set operators $\cup$ (union), $\cap$ (intersect), $-$ (set difference), and an overline $\overline{x}$ as complement operator on sets. Section 3.2 defines constraints and their violations. Schemas are treated in section 3.3. Then section 3.4 defines information systems.

3.1 Data sets

A data set $\mathscr{D}$ describes a set of structured data, which is typically stored persistently in a database of some kind. The notation $\mathscr{D}_{\mathscr{S}}$ refers to the data set of a particular system $\mathscr{S}$. The purpose of a data set is to describe the data of a system at one point in time. Before defining data sets, let us first define the constituent notions of atom, concept, relation, and triple.

Atoms serve as data elements. All atoms are taken from a set called $\mathbb{A}$. Concepts (concept symbols) can be understood as (names for) types, and our definitions do not exclude sub-typing. All concept symbols are taken from a set $\mathbb{C}$. For example, a developer might choose to classify `Peter` and `Melissa` as `Person`, and `074238991` as a `TelephoneNumber`. In this example, `Person` and `TelephoneNumber` are concept symbols. Moreover, `Peter`, `Melissa` and `074238991` are atoms. In the sequel, variables A, B, C, D will represent concept symbols, and variables a, b, and c represent *atoms*. The relation $inst : \mathbb{A} \times \mathbb{C}$ relates atoms to concept symbols. The term $a\ inst\ C$ means that atom a is an *instance* of a concept denoted by C.

Relations serve to organize and store data, allowing developers to represent facts. In this paper, variables r, s, and d represent relation symbols. All relation symbols are taken from a set $\mathbb{R}$. $\mathbb{R}$ is disjoint from $\mathbb{C}$ and $\mathbb{A}$. Every relation symbol r has a name, a source concept, and a target concept. The notation $r = n_{\langle A,B\rangle}$ denotes that relation symbol r has name n, source concept A, and target concept B. The part $\langle A, B\rangle$ is called the *signature* of the relation symbol. When the signature is clear from the context, or not important, we use r and n interchangeably.

Triples serve to represent data. A triple is an element of $\mathbb{A} \times \mathbb{R} \times \mathbb{A}$. For example, $\langle \texttt{Peter}, phone_{\langle \texttt{Person},\texttt{TelephoneNumber}\rangle}, \texttt{074238991}\rangle$ is a triple.

Definition 1 (Data set). *A data set $\mathscr{D}$ is a tuple $\langle \mathcal{T}, inst\rangle$ with finite $\mathcal{T} \subseteq \mathbb{A} \times \mathbb{R} \times \mathbb{A}$ and $inst \subseteq \mathbb{A} \times \mathbb{C}$ that satisfies:*

$$\langle a, n_{\langle A,B\rangle}, b\rangle \in \mathcal{T} \Rightarrow a\ inst\ A \ \wedge\ b\ inst\ B \tag{1}$$

Looking at the example, equation 1 says that `Peter` is an instance of `Person` and `074238991` is an instance of `TelephoneNumber`. In practice, users can say that the person Peter has telephone number 074238991. So, the "thing" that `Peter` refers to (which is Peter) has `074238991` as a telephone number. The notations $\mathcal{T}_{\mathscr{D}}$ and $inst_{\mathscr{D}}$ are used to disambiguate $\mathcal{T}$ and $inst$ when necessary. To save writing in the sequel, the notation $a\ r\ b$ means that $\langle a, r, b\rangle \in \mathcal{T}$. We'll use $\mathbb{D}$ to denote the set of all possible data sets.

A relation symbol r can serve as a container of pairs, as defined by the function $pop_r : \mathbb{D} \to \mathcal{P}\{\mathbb{A} \times \mathbb{A}\}$. It defines a set of pairs, also known as the population of r:

$$pop_r(\mathscr{D}) \ = \ \{\langle a, b\rangle \mid \ \langle a, r, b\rangle \in \mathcal{T}_{\mathscr{D}}\} \tag{2}$$

Note that the phrase "pair $\langle a, b\rangle$ is in relation r" means that $\langle a, b\rangle \in pop_r(\mathscr{D})$ where $\mathscr{D}$ is clear from the context. We overload the notation pop so we can use it on concept symbols $pop_C : \mathbb{D} \to \mathcal{P}\{\mathbb{A}\}$ and expressions. We also define the difference of populations, in equation 4, both for relation and concept symbols:

$$pop_C(\mathscr{D}) = \{x \mid \ x \ \ inst_{\mathscr{D}} \ \ C\} \tag{3}$$

$$pop_{x-y}(\mathscr{D}) = pop_x(\mathscr{D}) - pop_y(\mathscr{D}) \tag{4}$$

3.2 Constraints

Every constraint symbol is an element of a set called $\mathbb{U}$. In this paper, variables u and v represent symbols for all three types of constraints. For every constraint u, function $viol_u : \mathbb{D} \to \mathcal{P}\{\mathbb{A} \times \mathbb{A}\}$ produces the violations of u, and $sign_u : \mathbb{C} \times \mathbb{C}$ yields the signature of u. The definition of $viol_u$ implies the assumption that we represent each violation as a pair. Every constraint must satisfy:

$$\langle a, b\rangle \in viol_u(\mathscr{D}) \ \wedge \ sign_u = \langle A, B\rangle \ \Rightarrow \ a \ inst \ A \ \wedge \ b \ inst \ B \tag{5}$$

Note that $viol_u(\mathscr{D}) = \emptyset$ means that $\mathscr{D}$ satisfies the constraint whose symbol is u. We'll say that u is satisfied in such cases.

In order to guarantee that the work required for migration is finite, it suffices to require that $viol_u(\mathscr{D})$ is a finite set. The language Ampersand implements many of the sets described so far as finite sets, causing any constraint that can be specified in it to satisfy that $viol_u(\mathscr{D})$ is finite.

In the current paper, we will, for the sake of simplicity, only consider transactional invariants for which violations can be repaired by inserting them into a single designated relation. The language Ampersand has more types of transactional invariants than just this one, but this is sufficient for this paper.

In case u denotes a transactional invariant, the system will keep it satisfied by adding the violations to a specific relation denoted by $n_{\langle A,B\rangle}$ such that $\langle A, B\rangle = sign_u$. This requires that adding the pair to that relation solves the violation:

$$\begin{aligned}&(a, b) \in viol_u(\langle \mathcal{T}, inst\rangle) \implies \\ &\quad viol_u(\langle \mathcal{T} \cup \{\langle a, n_{\langle A,B\rangle}, b\rangle\}, inst\rangle) = viol_u(\langle \mathcal{T}, inst\rangle) - \{(a, b)\}\end{aligned} \tag{6}$$

It is obvious that not every conceivable constraint can satisfy this equation. So, we assume that the compiler restricts the set of transactional invariants to those that satisfy equation 6. As $n_{\langle A,B\rangle}$ is specific for u, we can write $\texttt{enforce}_u$ for it. We call this the symbol for the *enforced relation* of the transactional invariant denoted by u:

$$\texttt{enforce}_u = n_{\langle A,B\rangle} \tag{7}$$

Let us denote a transactional invariant as $r \hookleftarrow viol_u$ or equivalently $r \hookleftarrow \lambda\mathscr{D}.\ viol_u(\mathscr{D})$, in which $r = \texttt{enforce}_u$. The symbol $u \in \mathbb{U}$ that refers to $\texttt{enforce}_u \hookleftarrow viol_u$ is written as $[\texttt{enforce}_u \hookleftarrow viol_u]$.

3.3 Schemas

Schemas serve to capture the semantics of a system [15]. They define concepts, relations, and constraints. We assume that a software engineer defines a schema on design time, and a compiler checks whether the semantics are consistent. Errors the compiler detects are prohibitive for generating code, to prevent a substantial class of mistakes to ever reach end-users.

We describe a schema $\mathscr{Z}$ as a tuple $\langle \mathcal{C}, \mathcal{R}, \mathcal{U}, \mathcal{E}, \mathcal{B} \rangle$, in which $\mathcal{C} \subseteq \mathbb{C}$ is a finite set of concept symbols, $\mathcal{R} \subseteq \mathbb{R}$ is a finite set of relation symbols, $\mathcal{U} \subseteq \mathbb{U}$ is a finite set of symbols for blocking invariants, $\mathcal{E} \subseteq \mathbb{U}$ is a finite set of symbols denoting transactional invariants, and $\mathcal{B} \subseteq \mathbb{U}$ is a finite set of symbols for business constraints.

Definition 2 (Schema). *A schema is a tuple $\langle \mathcal{C}, \mathcal{R}, \mathcal{U}, \mathcal{E}, \mathcal{B} \rangle$ that satisfies:*

$$n_{\langle A,B \rangle} \in \mathcal{R} \;\Rightarrow\; A \in \mathcal{C} \wedge B \in \mathcal{C} \tag{8}$$

$$u \in \mathcal{U} \cup \mathcal{E} \cup \mathcal{B} \;\wedge\; sign_u = \langle A, B \rangle \;\Rightarrow\; A \in \mathcal{C} \wedge B \in \mathcal{C} \tag{9}$$

$$u \in \mathcal{E} \;\Rightarrow\; \texttt{enforce}_u \in \mathcal{R} \tag{10}$$

Requirements 8 and 9 ensure that concept symbols mentioned in relations and in the signature of constraints are part of the schema. Requirement 10 ensures the enforced relation symbol of a transactional invariant is declared in the schema. When clarity is needed, we write $\mathcal{C}_{\mathscr{Z}}$, $\mathcal{R}_{\mathscr{Z}}$, $\mathcal{U}_{\mathscr{Z}}$, $\mathcal{E}_{\mathscr{Z}}$, $\mathcal{B}_{\mathscr{Z}}$ for $\mathcal{C}$, $\mathcal{R}$, $\mathcal{U}$, $\mathcal{E}$, $\mathcal{B}$ corresponding to $\mathscr{Z} = \langle \mathcal{C}, \mathcal{R}, \mathcal{U}, \mathcal{E}, \mathcal{B} \rangle$.

3.4 Information Systems

Let us now define information systems by enumerating the requirements.

Definition 3 (information system).
An information system $\mathscr{S}$ is a tuple $\langle \mathscr{D}, \mathscr{Z} \rangle$, in which

- *$\mathscr{D} = \langle \mathcal{T}, inst \rangle$ is a data set (so it must satisfy equation 1). We write $\mathcal{T}_{\mathscr{S}} = \mathcal{T}$ and $inst_{\mathscr{S}} = inst$ if needed;*
- *$\mathscr{Z} = \langle \mathcal{C}, \mathcal{R}, \mathcal{U}, \mathcal{E}, \mathcal{B} \rangle$ is a schema (so it must satisfy equations 8 thru 10).*
- *Triples in the data set must have their relation symbol mentioned in the schema:*

$$\langle a, n_{\langle A,B \rangle}, b \rangle \in \mathcal{T} \Rightarrow n_{\langle A,B \rangle} \in \mathcal{R} \tag{11}$$

- *All violations must have a type, which follows from (5).*
- *The system keeps any transactional invariant denoted by u satisfied by adding violations to the relation denoted by $\texttt{enforce}_u$ (6).*
- *All invariants must remain satisfied:*

$$\forall u \in \mathcal{U} \cup \mathcal{E}.\; viol_u(\mathscr{D}) = \emptyset \tag{12}$$

We assume that a deployment will fail if these requirements are not satisfied.

4 Generating a Migration Script

The complexity of migrating data to date yields expensive and error-prone migration projects. By generating the migration system we can prevent many human induced errors. However, to allow for human tailoring, we generate a script that describes this migration system (from which the system can be generated automatically).

This section starts with a presentation of the migration script that is to be used.

4.1 Generating a migration script

In the migration system, we need to refer to the items (concepts, relations, and constraints) of both the existing system and the desired system. We have to relabel items with prefixes to avoid name clashes in the migration system. We use a left arrow to denote relabeling by prefixing the name of the item with "old." (or some other prefix that avoids name clashes).

$$\begin{aligned}
\overleftarrow{\langle \mathscr{D}, \mathscr{X}\rangle} &= \langle \overleftarrow{\mathscr{D}}, \overleftarrow{\mathscr{X}}\rangle \\
\overleftarrow{\langle \mathcal{T}, inst\rangle} &= \langle \overleftarrow{\mathcal{T}}, inst\rangle \\
\overleftarrow{\langle \mathcal{C},\mathcal{R},\mathcal{U},\mathcal{E},\mathcal{B}\rangle} &= \langle \mathcal{C}, \overleftarrow{\mathcal{R}}, \overleftarrow{\mathcal{U}}, \overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{B}}\rangle \\
\overleftarrow{\mathcal{T}} &= \{\langle a, \overleftarrow{r}, b\rangle \mid \langle a,r,b\rangle \in \mathcal{T}\} \\
\overleftarrow{n_{\langle A,B\rangle}} &= old.n_{\langle A,B\rangle} \\
\overleftarrow{X} &= \{\overleftarrow{x} \mid x \in X\} \\
viol_{\overleftarrow{u}}(\overleftarrow{\mathscr{D}}) &= viol_u(\mathscr{D}) \\
sign_{\overleftarrow{u}} &= sign_u \\
\mathtt{enforce}_{\overleftarrow{u}} &= \overleftarrow{\mathtt{enforce}_u}
\end{aligned} \tag{13}$$

The notation $\overrightarrow{\mathscr{S}}$ is defined similarly, using prefix "new." instead of "old.". This relabeling does not have any effect on the behavior of the system, i.e. $\mathscr{S}$, $\overleftarrow{\mathscr{S}}$, and $\overrightarrow{\mathscr{S}}$ are indistinguishable for end-users.

Then we define the migration system $\mathscr{M}$ as follows: Let $\langle \mathscr{D}, \mathscr{X}\rangle$ be the existing system. Let $\langle \mathscr{D}', \mathscr{X}'\rangle$ be the desired system in its initial state.

1. We take a disjoint union of the data sets by relabeling relation symbols, so the migration script can refer to relations from both systems:

$$\mathscr{D}_{\mathscr{M}} = \overleftarrow{\mathscr{D}} \cup \overrightarrow{\mathscr{D}'} \tag{14}$$

2. We create transactional invariants to copy the population of relations from $\mathscr{D}$ to $\mathscr{D}'$: For every relation r shared by the existing and desired systems, we generate a helper relation: $\mathtt{copy}_r$, and two transactional invariants. The first transactional invariant populates relation $\overrightarrow{r}$ and the second populates $\mathtt{copy}_r$. The helper relation $\mathtt{copy}_r$ contains the pairs that have been copied. We use

the helper relation to keep the transactional invariants from immediately repopulating $\overrightarrow{r}$ when a user deletes triples in $\mathscr{D}'$.

$$\mathcal{R}_1 = \{\texttt{copy}_r \mid r \in \mathcal{R}_{\mathscr{Z}'} \cap \mathcal{R}_{\mathscr{Z}}\} \tag{15}$$

$$\begin{aligned}\mathcal{E}_1 = &\left\{\left[\overrightarrow{r} \hookleftarrow \mathit{pop}_{\overleftarrow{r}-\texttt{copy}_r}\right] \,\middle|\, r \in \mathcal{R}_{\mathscr{Z}'} \cap \mathcal{R}_{\mathscr{Z}}\right\} \cup \\ &\{[\texttt{copy}_r \hookleftarrow \mathit{pop}_{\overrightarrow{r}\cap\overleftarrow{r}}] \mid r \in \mathcal{R}_{\mathscr{Z}'} \cap \mathcal{R}_{\mathscr{Z}}\}\end{aligned} \tag{16}$$

The copying process terminates when:

$$\forall r \in \mathcal{R}_{\mathscr{Z}'} \cap \mathcal{R}_{\mathscr{Z}}.\ \overleftarrow{r} = \texttt{copy}_r \tag{17}$$

Since the enforce rules only insert triples, the copying process is guaranteed to terminate. However, deletions in the old system that happen during this copying process might not propagate into the migration system. This may pose a risk to the business with respect to data quality.

3. The new blocking invariants are $\mathcal{U}_{\mathscr{Z}'} - \mathcal{U}_{\mathscr{Z}}$. For each new blocking invariant u, we generate a helper relation: $\texttt{fixed}_u$, to register all violations that are fixed, and a blocking invariant v in the migration system that blocks fixed violations from recurring:

$$\mathcal{R}_2 = \{\texttt{fixed}_u \mid u \in \overrightarrow{\mathcal{U}_{\mathscr{Z}'} - \mathcal{U}_{\mathscr{Z}}}\} \tag{18}$$

$$\begin{aligned}\mathcal{U}_{\text{block}} = \{v\ &\textbf{with} \\ &\mathit{sign}_v = \mathit{sign}_u \\ &\mathit{viol}_v(\mathscr{D}) = \mathit{viol}_u(\mathscr{D}) \cap \mathit{pop}_{\texttt{fixed}_u}(\mathscr{D}) \\ \mid u \in &\overrightarrow{\mathcal{U}_{\mathscr{Z}'} - \mathcal{U}_{\mathscr{Z}}}\}\end{aligned} \tag{19}$$

4. We use a transactional invariant to produce the population of the helper relation $\texttt{fixed}_u$.

$$\mathcal{E}_2 \;=\; \{\texttt{fixed}_u \hookleftarrow \lambda\mathscr{D}.\ \overline{\mathit{viol}_u(\mathscr{D}) \cup \mathit{pop}_{\texttt{fixed}_u}(\mathscr{D})} \mid u \in \overrightarrow{\mathcal{U}_{\mathscr{Z}'} - \mathcal{U}_{\mathscr{Z}}}\} \tag{20}$$

5. To signal users that there are violations that need to be fixed, we generate a business constraint for each new blocking invariant denoted by u:

$$\begin{aligned}\mathcal{B}_{\text{fix}} = \{v\ &\textbf{with} \\ &\mathit{sign}_v = \mathit{sign}_u \\ &\mathit{viol}_v(\mathscr{D}) = \mathit{viol}_u(\mathscr{D}) \\ \mid u \in &\overrightarrow{\mathcal{U}_{\mathscr{Z}'} - \mathcal{U}_{\mathscr{Z}}}\}\end{aligned} \tag{21}$$

In some cases, a migration engineer can invent ways to satisfy these invariants automatically. This is one of the places where it is useful for the generator to produce source code (as opposed to compiled code) to allow the migration engineer to replace a business constraint with transactional invariants of her own making. After all violations are fixed, i.e. when equation 22 is satisfied, the migration engineer can switch the ingress to the desired system. This occurs at MoC and replaces $\mathcal{U}_{\text{block}}$ in the migration system by the blocking invariants of the desired system. This moment arrives when:

$$\forall u \in \overrightarrow{\mathcal{U}_{\mathscr{Z}'} - \mathcal{U}_{\mathscr{Z}}}.\ viol_u(\mathscr{D}) \subseteq pop_{\mathtt{fixed}_u}(\mathscr{D}) \tag{22}$$

Equivalently, $\forall u \in \overrightarrow{\mathcal{U}_{\mathscr{Z}'} - \mathcal{U}_{\mathscr{Z}}}.\ viol_u(\mathscr{D}) = \emptyset$. After this, the migration engineer can remove the migration system and the old system.

6. Let us combine the above into a single migration schema:

$$\begin{aligned}\mathscr{Z}_{\mathscr{M}} = \langle & \mathcal{C}_{\mathscr{D}} \cup \mathcal{C}_{\mathscr{D}'}, \\ & \overleftarrow{\mathcal{R}_{\mathscr{Z}}} \cup \overrightarrow{\mathcal{R}_{\mathscr{Z}'}} \cup \mathcal{R}_1 \cup \mathcal{R}_2, \\ & \mathcal{U}_{\text{block}} \cup \overrightarrow{\mathcal{U}_{\mathscr{Z}} \cap \mathcal{U}_{\mathscr{Z}'}}, \\ & \mathcal{E}_1 \cup \mathcal{E}_2 \cup \overrightarrow{\mathcal{E}_{\mathscr{Z}'}}, \\ & \mathcal{B}_{\text{fix}} \cup \overrightarrow{\mathcal{B}_{\mathscr{Z}'}} \rangle\end{aligned} \tag{23}$$

This schema represents the migration system. In our reasoning, we have only used information from the schemas of the existing system and the desired system. This shows that it can be generated from these schemas without using any knowledge of the data sets.

5 Proof of Concept

By way of proof of concept (PoC), we have built a migration system in Ampersand. To demonstrate it in the context of this paper, the existing system, $\langle \mathscr{D}, \mathscr{Z} \rangle$, is rather trivial. It has no constraints and just one relation, $r_{\langle A,B \rangle}$. Its population is $A = \{a_1, a_2, a_3\}$, $B = \{b_1\}$, and $pop_r(\mathscr{D}) = \{\langle a_1, b_1 \rangle\}$. The desired system contains one blocking invariant, which is the totality of $r_{\langle A,B \rangle}$. Its violations are $\langle a_2, a_2 \rangle$ and $\langle a_3, a_3 \rangle$. The schema of the migration system, $\mathscr{Z}_{\mathscr{M}}$, follows from definition 23.

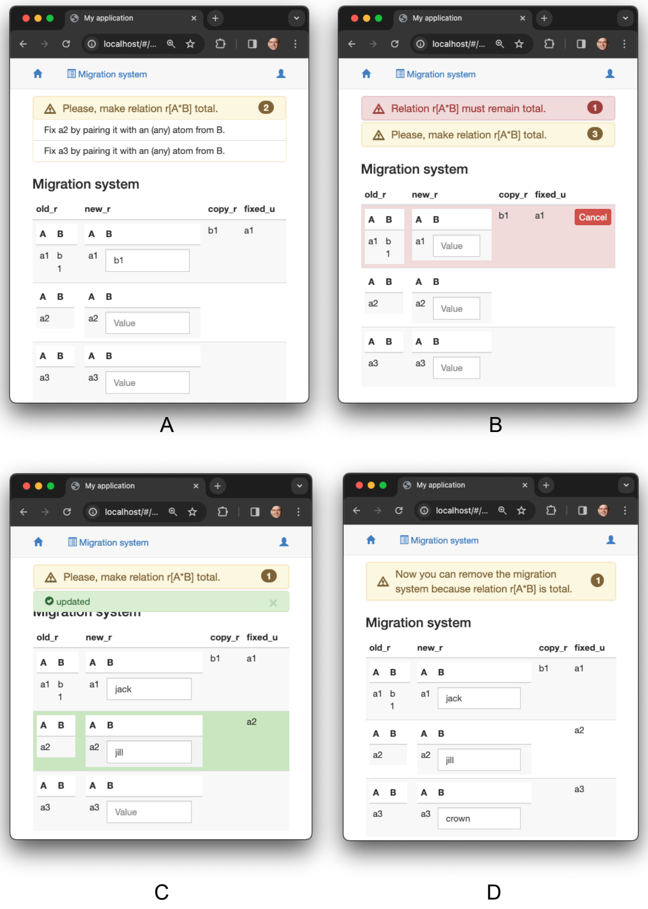
Figure 3 shows four successive screenshots, featuring $\overleftarrow{r}$ as `old_r`, $\overrightarrow{r}$ as `new_r`.

Exhibit A shows the migration system just after deployment, at the MoT. It shows that the copying of `old_r` to `new_r` has worked. The yellow message in exhibit A indicates that a user needs to fix totality for a_2 and a_3. The column `fixed_u` contains the elements for which totality is satisfied, so these fields are blocked from becoming empty.

Exhibit B shows an attempt to remove $\langle a_1, b_1 \rangle$ from `new_r` The red message blocks this from happening and the user gets a "Cancel" button to roll back the action. Note that the fields for a_2 and a_3 are empty, which is fine because they will not be blocked until they are given some value.

Exhibit C shows that the user fills in "Jill", which means that $\langle a_1, Jill \rangle$ is added to `new_r`.

Exhibit D: When the last atom of `A` is paired with an atom from `B`, requirement 22 is satisfied and the prototype informs the user to remove the migration system.



A

⚠ Please, make relation r[A*B] total. 2

Fix a2 by pairing it with an (any) atom from B.

Fix a3 by pairing it with an (any) atom from B.

Migration system

old_r	new_r	copy_r	fixed_u
A B a1 b1	A B a1 b1	b1	a1
A B a2	A B a2 Value		
A B a3	A B a3 Value		

B

⚠ Relation r[A*B] must remain total. 1

⚠ Please, make relation r[A*B] total. 3

Migration system

old_r	new_r	copy_r	fixed_u	
A B a1 b1	A B a1 Value	b1	a1	Cancel
A B a2	A B a2 Value			
A B a3	A B a3 Value			

C

⚠ Please, make relation r[A*B] total. 1

✓ updated ×

Migration system

old_r	new_r	copy_r	fixed_u
A B a1 b1	A B a1 jack	b1	a1
A B a2	A B a2 jill		a2
A B a3	A B a3 Value		

D

⚠ Now you can remove the migration system because relation r[A*B] is total. 1

Migration system

old_r	new_r	copy_r	fixed_u
A B a1 b1	A B a1 jack	b1	a1
A B a2	A B a2 jill		a2
A B a3	A B a3 crown		a3

Fig. 3. Four successive screenshots in the PoC

5.1 Validation

The proof of concept gives but one example of something that works. In fact, having built prototypes increases our confidence, but cannot serve as a proof. This section argues the validity of our method. We take advantage of the formal definition of the generated migration system (section 4) to precisely state the assumptions and requirements of its validity.

The initial situation consists of one existing system $\mathscr{S}$ and one desired system $\mathscr{S}'$ of which we have a schema, $\mathscr{Z}_{\mathscr{S}'}$ and an initial dataset, $\mathscr{D}_{\mathscr{S}'}$. We may assume that $\mathscr{S}$ satisfies definition 3 because it is a system in production. Also, $\mathscr{Z}_{\mathscr{S}'}$ satisfies equations 8 thru 10 because the schema of the desired system is type-correct. Together, the schema and the intial dataset forms the desired system

$\langle \mathscr{Z}_{\mathscr{S}'}, \mathscr{D}_{\mathscr{S}'} \rangle$, which satisfies definition 3. With these assumptions in place, we must verify that:

1. After predeployment, the migration system $\mathscr{M}$ copies the designated triples to the desired system in finite time.
2. At MoT, $\mathscr{M}$ satisfies the definition of information system (3), especially that it has no violations of blocking invariants (requirement 12) to ensure a successful deployment.
3. Once business actors have fixed the new business invariants, i.e. when the condition for the MoC (requirement 22) is satisfied, the behavior of $\mathscr{M}$ and the desired system is identical, so moving the ingress from $\mathscr{M}$ to the desired system is not noticable for end-users.
4. The old system has become redundant, so we can remove $\mathscr{S}$ and $\mathscr{M}$.

Let us discuss these points one by one.

The relations to be copied from $\mathscr{S}$ are those relations that the desired system retains: $\mathcal{R}_{\mathscr{Z}'} \cap \mathcal{R}_{\mathscr{Z}}$. For each r to be copied from $\mathscr{S}$, $\mathscr{M}$ contains a relation symbol $\mathtt{copy}_r$ in $\mathcal{R}_1$ (eqn. 15). After the MoT, the ingress sends all change events to $\mathscr{M}$, so the existing system can finish the work it is doing for transactional invariants and will not change after that. In other words, the population of every relation symbol in $\mathcal{R}_{\mathscr{Z}}$ becomes stable and so does every $\mathtt{copy}_r$. At that point in time, eqn. 17 is satisfied and stays satisfied. Effectively, $\mathcal{E}_1$ becomes redundant once the copying is done.

Then, $\mathscr{M}$ contains only blocking invariants that exist in the existing system as well (def. 2.3). For this reason, all of these blocking invariants are satisfied on MoT. Since $\mathscr{M}$ contains no other blocking invariants, it satisfies requirement 12. This implies that $\mathscr{M}$ works and the migration engineer may safely switch the ingress from $\mathscr{S}$ to $\mathscr{M}$.

Thirdly, the constraints that may need human intervention are the blocking invariants of the desired system that were not in the existing system ($\mathcal{U}_{\mathscr{Z}'} - \mathcal{U}_{\mathscr{Z}}$ in def. 21). $\mathscr{M}$ features $\mathcal{B}_{\text{fix}}$ to represent these invariants in the guise of business constraints. This lets business actors resolve the violations. Each violation in $\mathcal{B}_{\text{fix}}$ that a business actor resolves, will never reappear because it is registered in $\mathtt{fixed}_r$ by the transactional invariants of $\mathcal{E}_2$ (eqn. 20). When all violations are fixed, every rule in $\mathcal{U}_{\text{block}}$ has become blocking. So, after the MoC, $\mathcal{U}_{\text{block}} \cup \overrightarrow{\mathcal{U}_{\mathscr{Z}} \cap \mathcal{U}_{\mathscr{Z}'}}$ equals $\overrightarrow{\mathcal{U}_{\mathscr{Z}'}}$ in the migration system.

The constraints in the desired system are partly written to resolve data pollution. In some cases, the migration engineer wants users to get rid of that pollution and turn the rule in a blocking invariant as described above. However, some of these constraints might be satisfiable automatically. In that case, the migration engineer might substitute such constraints from $\mathcal{B}_{\text{fix}}$ by transactional invariants in $\mathcal{E}_{\mathscr{M}}$. These invariants don't need the mechanism described above, because the migration system itself will take care that all constraints in $\mathcal{E}_{\mathscr{M}}$ are satisfied. Both cases need to be resolved at MoC.

So finally, when all violations are resolved, the constraints in $\mathcal{B}_{\text{fix}}$ have effectively become blocking invariants. The blocking invariants in the desired system consist of $\mathcal{U}_{\mathscr{Z}}$ and $\mathcal{B}_{\text{fix}}$, which is equivalent to $\mathcal{U}_{\mathscr{Z}'}$. Hence we can replace $\mathcal{B}_{\text{fix}} \cup \mathcal{U}_{\mathscr{Z}}$ with $\mathcal{U}_{\mathscr{Z}'}$ after condition 22 is satisfied.

Now we can assemble the results at MoC. The above reasoning shows that at MoC $\langle \mathcal{C}_{\mathscr{D}} \cup \mathcal{C}_{\mathscr{D}'}, \overleftarrow{\mathcal{R}_{\mathscr{Z}}} \cup \overrightarrow{\mathcal{R}_{\mathscr{Z}'}} \cup \mathcal{R}_1 \cup \mathcal{R}_2, \mathcal{U}_{\text{block}} \cup \overrightarrow{\mathcal{U}_{\mathscr{Z}} \cap \mathcal{U}_{\mathscr{Z}'}}, \mathcal{E}_1 \cup \mathcal{E}_2 \cup \overrightarrow{\mathcal{E}_{\mathscr{Z}'}}, \mathcal{B}_{\text{fix}} \cup \overrightarrow{\mathcal{B}_{\mathscr{Z}'}} \rangle$ is equivalent to $\langle \mathcal{C}_{\mathscr{D}'}, \overrightarrow{\mathcal{R}_{\mathscr{Z}'}}, \overrightarrow{\mathcal{U}_{\mathscr{Z}'}}, \overrightarrow{\mathcal{E}_{\mathscr{Z}'}}, \overrightarrow{\mathcal{B}_{\mathscr{Z}'}} \rangle$, which is equal to $\overrightarrow{\mathscr{Z}'}$. So from MoC onwards, $\langle \mathscr{D}_{\mathscr{M}}, \mathscr{Z}_{\mathscr{M}} \rangle$ is equivalent to $\langle \mathscr{D}_{\mathscr{M}}, \overrightarrow{\mathscr{Z}'} \rangle$. Hence, we can tell that $\mathscr{D}_{\mathscr{M}}$ is a valid dataset for the desired system, so we can switch the ingress from the migration system to the desired system without users noticing the difference. Then, the migration system gets no more inputs, so it can be removed. Since $\mathcal{E}_1$ and $\mathcal{E}_2$ are redundant after the MoC, we can retain $\overrightarrow{\mathcal{E}_{\mathscr{Z}'}}$ in $\mathscr{M}$, which is equivalent to $\mathcal{E}_{\mathscr{Z}'}$ in the desired system. Likewise, $\mathcal{B}_{\text{fix}}$ has become redundant, so $\mathscr{M}$ can do with just $\overrightarrow{\mathcal{B}_{\mathscr{Z}'}}$. In the desired system, that is equivalent to $\mathcal{B}_{\mathscr{Z}'}$. So, the constraints in the desired system after the MoC are equivalent to the constraints in the MoC.

6 Conclusions

In this paper, we describe the data migration as going from an existing system to a desired one, where the schema changes. As Ampersand generates information systems, creating a new system can be a small task, allowing for incremental deployment of new features. We describe the parts of a system that have an effect on data pollution. We assume that the existing system does not violate any constraints of its schema, but address other forms of data pollution: constraints that are not in the schema but are in the desired schema are initially relaxed such that the business can start using the migration system, after which this form of data pollution needs to be addressed by human intervention. We propose a method for doing migration such that only a finite amount of human intervention is needed. Our method allows a system similar to the desired system to be used while the intervention takes place.

Our proposed migration is certainly not the only approach one could think of. However, we have not come across other approaches that allow changing the schema in the presence of constraints. As such, we cannot compare our approach against other approaches. We envision that one day there will be multiple approaches for migration under a changing schema to choose from. For now, our next step is to automate the generation of migration scripts as an extension to Ampersand.

This work does not consider what to do about (user) interfaces. Instead, it models events by assuming that any change to the data set can be achieved. In practice, such changes need to be achieved through interfaces. Most Ampersand systems indeed allow the users of the system to edit the data set quite freely through the interfaces. However, some interfaces may require certain constraints to be satisfied, which means that interfaces of the desired system might break when used through the migration system. In the spirit of the approach outlined here, we hope to generate migration interfaces that can replace any broken interfaces until the Moment of Transition. How to do this is future work.

References

1. Ataei, P., Khan, F., Walkingshaw, E.: A variational database management system. In: Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. pp. 29–42. GPCE 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3486609.3487197
2. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. SEI Series in Software Engineering, Addison-Wesley, New York (2015). https://doi.org/10.5555/2810087
3. Gholami, M.F., Daneshgar, F., Low, G., Beydoun, G.: Cloud migration process — a survey, evaluation framework, and open challenges. J. Syst. Softw. **120**(C), 31–69 (oct 2016). https://doi.org/10.1016/j.jss.2016.06.068
4. Hanenberg, S., Kleinschmager, S., Robbes, R., Tanter, É., Stefik, A.: An Empirical Study on the Impact of Static Typing on Software Maintainability. Empirical Software Engineering **19**(5), 1335–1382 (2014). https://doi.org/10.1007/s10664-013-9289-1
5. Hattensperger, C., Kempf, P.: Towards a Formal Framework for Heterogeneous Relation Algebra. Information Sciences: an International Journal **119**, 193–203 (October 1999). https://doi.org/10.1016/S0020-0255(99)00014-6
6. Hillenbrand, A., Störl, U., Nabiyev, S., Klettke, M.: Self-adapting data migration in the context of schema evolution in NoSQL databases. Distributed and Parallel Databases **40**(1), 5–25 (2022)
7. Humanitec: Devops benchmarking study 2023. Tech. rep., Humanitec Inc., Berlin, New York (2023)
8. IEEE: Architecture Working Group of the Software Engineering Committee: Standard 1471-2000: Recommended Practice for Architectural Description of Software Intensive Systems. IEEE Standards Department (2000)
9. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
10. Joosten, S.: Software development in relation algebra with Ampersand. In: Höfner, P., Pous, D., Struth, G. (eds.) Relational and Algebraic Methods in Computer Science: 16th International Conference, RAMiCS 2017, Lyon, France, May 15-18, 2017, Proceedings. pp. 177–192. Springer International Publishing, Cham (2017)
11. Joosten, S.: Relation algebra as programming language using the Ampersand compiler. Journal of Logical and Algebraic Methods in Programming **100**, 113–129 (2018). https://doi.org/10.1016/j.jlamp.2018.04.002
12. Joosten, S., Wedemeijer, L., Michels, G.: Rule-Based Design. Open Universiteit, Heerlen (2013)
13. Michels, G.: Development Environment for Rule-based Prototyping. Ph.D. thesis, Open University of the Netherlands (June 2015)
14. Petersen, P., Hanenberg, S., Robbes, R.: An Empirical Comparison of Static and Dynamic Type Systems on API Usage in the Presence of an IDE: Java vs. Groovy with Eclipse. In: Proceedings of the 22Nd International Conference on Program Comprehension. pp. 212–222. ICPC 2014, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2597008.2597152
15. Spivak, D.I.: Functorial data migration. Information and computation **217**, 31–51 (2012)
16. Theodorou, V., Abelló, A., Thiele, M., Lehner, W.: Frequent patterns in ETL workflows: An empirical approach. Data & Knowledge Engineering **112**, 1–16 (2017). https://doi.org/10.1016/j.datak.2017.08.004

17. Walkingshaw, E., Kästner, C., Erwig, M., Apel, S., Bodden, E.: Variational data structures: Exploring tradeoffs in computing with variability. In: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. pp. 213–226. Onward! 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2661136.2661143
18. Woude, J.v.d., Joosten, S.: Relational Heterogeneity Relaxed by Subtyping. In: Proceedings of the 12th conference on Relational and Algebraic Methods in Computer Science. pp. 347–361. Lecture Notes in Computer Science 6663, Springer-Verlag, Berlin (2011)
19. Wu, B., Grimson, J., Bisbal, J., Lawless, D.: Legacy information systems: Issues and directions. IEEE Software **16**(05), 103–111 (sep 1999). https://doi.org/10.1109/52.795108