On the Implementation and Correctness of Information System Upgrades

Klaus Haller
COMIT AG
Zürich, Switzerland
klaus.haller@comit.ch

Abstract— **Information systems are applications incorporating a database for storing and processing data. Upgrading information systems requires updating the application logic, modifying the database schema, and adopting the data accordingly. Previous research focuses either on schema evolution or on application logic updates. In this paper, we take a holistic approach by addressing the combination. First, we elaborate the three main upgrade patterns: install & copy, rejuvenation/delta only, and rejuvenation/verified. Second, we introduce our upgrade correctness concept. It is a formal correctness criterion for deciding whether an upgrade succeeded. Third, we discuss implementation patterns. Our insights base on various upgrade projects from stand-alone applications to multi-tenant systems having affected more than one hundred banks.**

Keywords— Information Systems, Database Applications, Upgrades, Correctness, Testing

I. INTRODUCTION

Most of today's business applications are information systems. Information system release upgrades pose special challenges due to the combination of "classical" release upgrade topics (e.g., software deployment) and schema evolution including data transformation.

Though Lämmel [1] elaborated the essence of upgrading systems with coupled components, the most influential research concentrates on the various and important open issues for systems with one component. The Clipper prototype [2], e.g., focuses on reconfiguring systems online without shutting them down. Malik and Hassan [3] discuss upgrade development. Jansen et al. [4] look at the interaction between a provider of new software and a client. The work of Rellermeyer et al. [5] is a typical approach to the challenges of upgrading distributed systems, as is the work of Ajmani et al. [6]. Oreizy et al. [7] take an architectural perspective. There is also much important work about schema evolution in the area of databases (e.g., Curion et al. [8] and Yu and Popa [9]) and about ERP upgrades (e.g., Beatty and Williams [10]). Biermann et al. [11] looked on theoretical issues and introduced a formal model for dynamic updates.

In this paper, we combine the database-centric perspective on schema evolution with the task of upgrading the application logic. We introduce our information system model in Section II. Section III presents our first contribution: the main upgrade patterns for deploying the new application logic and the adoption of data and tables during an upgrade. As our second contribution, we formalize information system upgrade correctness in Section IV, for which (to our best knowledge) only an intuitive understanding exists up to now. The third contribution, concrete implementation patterns, follows in Section V. Section VI concludes the paper with a short summary.

II. INFORMATION SYSTEM MODEL

Discussing upgrade implementation and correctness issues requires a clear understanding of the term **information system**. Throughout this paper, we understand an information system as a quintuple IS $= \langle \mathcal{A}, \mathcal{W}, \mathcal{T}, \mathcal{D}, \mathcal{P} \rangle$. $\mathcal{A}$ denotes the *application logic* including GUIs and output texts. It resides either in the database ($\mathcal{A}_{DB}$, e.g., stored procedures), outside ($\mathcal{A}_{OU}$, e.g., implemented using .NET or J2EE), or is spread over both (obviously, $\mathcal{A} = \mathcal{A}_{DB} \cup \mathcal{A}_{OU}$ holds). The *workflows* $\mathcal{W}$ are the interface to the users and define how to use the information system. The *tables* $\mathcal{T}$ in the database store the data. The *data* $\mathcal{D}$ falls into two groups: customer data $\mathcal{D}_{Cus}$ and vendor data $\mathcal{D}_{Ven}$ (with $\mathcal{D} = \mathcal{D}_{Ven} \cup \mathcal{D}_{Cus}$). Customer data is installation specific, e.g., customers of the bank running the system. Vendor data is shipped from the vendor to all customers. It is the same for all installations. An example would be a list of all European stock exchanges. Finally, *configuration parameters* $\mathcal{P}$ configure the interfaces to other services (e.g., the SWIFT network) or the system's behavior and appearance (e.g., date and time format).

Traditionally, information systems have been designed for or in one company. This changed with the emerge of standard software such as ERP systems. Now information systems are often products; i.e., they are installed by various customers. If we want to check whether an upgrade from version 2.1 to 3.0 is successful, we must check whether the upgraded information system behaves as a "newly installed" version 3.0 installation. We use the term **reference installation** for referring to such a "newly installed" version.

III. IMPLEMENTING UPGRADES

An upgrade implementation comprises two important elements:

- functionality for deploying the application logic and vendor data of the new version, and
- functionality for adopting tables and customer data such that they "fit" to the new application logic.

These elements are the *building blocks* for implementing upgrades. Upgrade patterns are *building plans*. They describe how to assemble the building blocks. They define when each

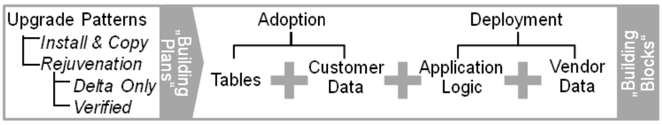

Figure 1: Release Upgrade Implementation Aspects

action is done and on which system. We discuss the upgrade patterns first, before we look at the adoption implementation. Figure 1 illustrates the different aspects and their interplay. Based on our experience in projects, we can assume that applications might be shut down for some hours during an upgrade.

A. Upgrade Patterns

Upgrade patterns structure the path from an "old" version with customer data to a "new" version, while keeping the customer data. Our first upgrade pattern is the *install & copy pattern*. In Step 1, a second information system with the new application logic, tables, and vendor data is installed (Figure 2). This second system does not have any customer data yet nor are any configuration parameters set. The latter one takes place in Step 2. Step 3 covers copying the customer data from the old to the new information system (certainly, this works only for a limited amount of data). The copying includes adopting the data if the old and new tables differ. Finally, the users (and interfaces of other systems) switch from the old to the new installation. The old system is shut down (Step 4).

		Step 1	*Step 2*	*Step 3*	*Step 4*
"Old"	$\mathcal{A}$	$\mathcal{A}^b$	$\mathcal{A}^b$	$\mathcal{A}^b$	*Shutdown*
	$\mathcal{T}$	$\mathcal{T}^b$	$\mathcal{T}^b$	$\mathcal{T}^b$	
	$\mathcal{D}$ Ven	$\mathcal{D}^b_{Ven}$	$\mathcal{D}^b_{Ven}$	$\mathcal{D}^b_{Ven}$	
	$\mathcal{D}$ Cus	$\mathcal{D}^b_{Cus}$	$\mathcal{D}^b_{Cus}$	$\mathcal{D}^b_{Cus}$	
	$\mathcal{P}$	$\mathcal{P}^b$	$\mathcal{P}^b$	$\mathcal{P}^b$	
"New"	$\mathcal{A}$	$\mathcal{A}^u$	$\mathcal{A}^u$	$\mathcal{A}^u$	$\mathcal{A}^u$
	$\mathcal{T}$	$\mathcal{T}^u$	$\mathcal{T}^u$	$\mathcal{T}^u$	$\mathcal{T}^u$
	$\mathcal{D}$ Ven	$\mathcal{D}^u_{Ven}$	$\mathcal{D}^u_{Ven}$	$\mathcal{D}^u_{Ven}$	$\mathcal{D}^u_{Ven}$
	$\mathcal{D}$ Cus			$\mathcal{D}^u_{Cus}$	$\mathcal{D}^u_{Cus}$
	$\mathcal{P}$		$\mathcal{P}^u$	$\mathcal{P}^u$	$\mathcal{P}^u$

Figure 2: Install & Copy Pattern (grey: system in use, black background: changed in this step; "b" means before the upgrade, "u" upgraded/new version)

An alternative pattern is the *rejuvenation pattern*. It "lifts" the existing information system to the higher version. The *rejuvenation/delta only* pattern adopts the tables, the customer data, and the vendor data in Step 1 (Figure 3). Thus, the "old" application logic becomes corrupt. It does not fit to the new tables anymore. In Step 2, the application logic is modified to become equal to the one of a reference installation, and, if necessary, the configuration parameters are adjusted. Now the system can be used again.

	Before	*Step 1*	*Step 2*	*After*
$\mathcal{A}$	$\mathcal{A}^b$	~~$\mathcal{A}^b$~~	$\mathcal{A}^u$	$\mathcal{A}^u$
$\mathcal{T}$	$\mathcal{T}^b$	$\mathcal{T}^u$	$\mathcal{T}^u$	$\mathcal{T}^u$
$\mathcal{D}$ Ven	$\mathcal{D}^b_{Ven}$	$\mathcal{D}^u_{Ven}$	$\mathcal{D}^u_{Ven}$	$\mathcal{D}^u_{Ven}$
$\mathcal{D}$ Cus	$\mathcal{D}^b_{Cus}$	$\mathcal{D}^u_{Cus}$	$\mathcal{D}^u_{Cus}$	$\mathcal{D}^u_{Cus}$
$\mathcal{P}$	$\mathcal{P}^b$	$\mathcal{P}^b$	$\mathcal{P}^u$	$\mathcal{P}^u$

Figure 3: Rejuvenation Pattern/Delta Only (strikethrough: application logic inconsistent with tables)

The second rejuvenation pattern is the *verified pattern* (Figure 4). It removes the application logic and vendor data in Step 1. It adopts the tables, the customer data, and the configuration parameters in Step 2. In Step 3, it deploys the new application logic and vendor data.

	Before	*Step 1*	*Step 2*	*Step 3*	*After*
$\mathcal{A}$	$\mathcal{A}^b$			$\mathcal{A}^u$	$\mathcal{A}^u$
$\mathcal{T}$	$\mathcal{T}^b$	$\mathcal{T}^b$	$\mathcal{T}^u$	$\mathcal{T}^u$	$\mathcal{T}^u$
$\mathcal{D}$ Ven	$\mathcal{D}^b_{Ven}$			$\mathcal{D}^u_{Ven}$	$\mathcal{D}^u_{Ven}$
$\mathcal{D}$ Cus	$\mathcal{D}^b_{Cus}$	$\mathcal{D}^b_{Cus}$	$\mathcal{D}^u_{Cus}$	$\mathcal{D}^u_{Cus}$	$\mathcal{D}^u_{Cus}$
$\mathcal{P}$	$\mathcal{P}^b$	$\mathcal{P}^b$	$\mathcal{P}^u$	$\mathcal{P}^u$	$\mathcal{P}^u$

Figure 4: Rejuvenation Pattern/Verified

The patterns have different characteristics as summarized in Table 1. The install & copy pattern incurs development costs for the vendor for copying and adopting the customer data. The customer has the (relatively low) costs for running the data copy functionality. However, the costs for setting-up a second system are often high. The rejuvenation patterns cause costs for developing the delta for the tables. The delta-only pattern causes extra costs for determining (and removing) the application logic delta between the old and the new version.

Variance and version consistency are relevant if (small) customer specific modifications exist as a result of the business model (see [12]). *Variance tolerance* reflects how well a pattern works for (slightly) modified installations. Incompatible tables are a problem for all patterns. Application logic modifications, however, do not pose problems for the install & copy and the rejuvenation/verified pattern. Both install always the complete new application logic from scratch. *Version consistency* looks at whether the application logic and tables are guaranteed to equal the reference installations after the upgrade. The install & copy pattern ensures this by a new installation of tables and application logic. The rejuvenation/verified installs the new application logic, but not the tables. So there are no guarantees for the tables. The rejuvenation/delta only gives no guarantees.

TABLE 1: COMPARISON UPGRADE PATTERNS ($\mathcal{A}$ APPLICATION LOGIC, $\mathcal{T}$ TABLES , ✓ GUARANTEED, ? NO GUARANTEE, QUALITY/COST RISK)

Pattern	**Costs**		**Variance Tolerance**		**Version Consistency**	
	Vendor	**Customer**	$\mathcal{T}$	$\mathcal{A}$	$\mathcal{T}$	$\mathcal{A}$
Install & Copy	Copy/Trans-formation $\mathcal{D}_{Cus}$	Execution, Set-Up	?	✓	✓	✓
Rejuvenation/ Delta Only	Delta for $\mathcal{T}$ and $\mathcal{A}$	Execution	?	?	?	?
Rejuvenation/ Verified	Delta for $\mathcal{T}$	Execution	?	✓	?	✓

B. Adopting Tables and Transforming Data

If the tables of the new versions are different, the upgrade must adopt the tables and the data in the tables. Each adoption has a change type and a hierarchy level. The **hierarchy level** defines the granularity. There are three for SQL databases: the schema level, the table level, and the attribute level. Four **change types** exist: "new/optional," "new/mandatory," "obsolete," and "modification." Two are easy to implement: "obsolete" by dropping the schema, table or attribute and "new/optional" by adding the new attribute, table, or schema. "Modification" changes the data model, e.g., one attribute is moved to another table. One copies the data to the new place before deleting the old data structure with its data. "New/mandatory" is more challenging. The data model is enhanced (e.g., by adding an attribute "nationality" to a customer table) and the new attribute is mandatory. Thus, we need values (data enrichment). The three main concepts are:

- A *default value* means that one value is used for all rows of a table, e.g., the nationality “Swiss.” Another option is “unknown.” Then, we could additionally demand that the first user opening the data in a GUI changes the value.
- In case of *user enrichment,* the user defines the values manually, e.g., using an Excel sheet with all customers for which we need a nationality. This sheet is loaded and used during the upgrade. It is a common approach in commercial projects, even for larger data sets, though the system downtime rises. The system must not be used any more after the data extraction for the Excel. Otherwise, new customers could be added for which no enrichment data is prepared.
- *Automatic enrichment* derives the missing data automatically, e.g., based on heuristics or data in other tables or databases.

The three data enrichment approaches, the four change types, and the granularity levels work for typical upgrade scenarios. For more complex situations, data migration research [13] provides additional concepts.

IV. Quality Assurance Concepts

When testing the correctness of an *application*, one executes different workflows. When testing the correctness of an *upgrade*, it is necessary – but not sufficient – that the upgrade itself executes correctly. Three more conditions must hold:

1. The tables of the upgraded information system must equal the ones of the reference installation.
2. The application logic must equal the one of the reference installation.
3. The semantics of the customer data must be unchanged.

Topics 1 and 2 are straight forward. Topic 3 is a challenge, because the data model can change (e.g., attributes might be moved between tables) or the semantics of the customer data can change even if data and tables do not change (e.g., when changing the system’s reference currency from $ to € one must recalculate all account balances). We refer to the semantics of our customer data by the term *interpretation*.

Definition (Interpretation): *The interpretation $\mathfrak{I}$ represents the semantics of all customer data. It depends on the table structure, the application logic, the vendor data, and the configuration parameters:* $\mathfrak{I}(\mathcal{D}_{cus}, \mathcal{T}, \mathcal{A}, \mathcal{D}_{Ven}, \mathcal{P})$

An upgrade is successful if, first, the interpretation of the customer data before and after the upgrade is the same. Second, the tables, the application logic, and the vendor data must equal the ones of the reference installation (Figure 5, ❶). Formally:

Definition (Upgrade Correctness): *Let* $is^b = \langle \mathcal{A}^b, \mathcal{T}^b, \mathcal{D}^b \rangle$ *with* $\mathcal{D}^b = \mathcal{D}^b_{Ven} \cup \mathcal{D}^b_{Cus}$ *be the information system before respectively* is^u *the information system after the upgrade. Let* is^{ref} *be the information system reference installation. The upgrade is correct if and only if:*

a) $\mathcal{A}^u = \mathcal{A}^{ref}$ *and* $\mathcal{T}^u = \mathcal{T}^{ref}$ *and* $\mathcal{D}^u_{Ven} = \mathcal{D}^{ref}_{Ven}$

b) $\mathfrak{I}(\mathcal{D}^u_{cus}, \mathcal{T}^u, \mathcal{A}^u, \mathcal{D}^u_{Ven}, \mathcal{P}^u) = \mathfrak{I}(\mathcal{D}^b_{cus}\, \mathcal{T}^b, \mathcal{A}^b, \mathcal{D}^b_{Ven}, \mathcal{P}^b)$

One could verify that the interpretation $\mathfrak{I}$ before and after the upgrade are the same by checking whether the *semantics* of *all data* are not changed by the upgrade. This is not possible in practice. A bank cannot compare the meaning of all data, i.e., of all customers, accounts, bookings, etc., after an information system upgrade. However, we can check whether *all data* is there. We can check separately whether the *semantics* of typical data items are the same as before the upgrade. If both checks succeed, we *assume* the semantics of all data items to be unchanged (Figure 5, ❷, “Upgrade Correctness Assumption”).

Choosing typical data items requires three steps. First, we separate relevant and irrelevant data. Logs are an example for data not having to be kept by an upgrade. Second, similar data items are grouped into business object testing classes (BOTC). They are defined based on a tester's perspective, but can map to object types or tables. Also, BOTCs can overlap. There might be, e.g., a BOTC for US nationals and one for wealthy customers. A wealthy US customer belongs to both BOTCs. Third, we choose one (or a few) typical data items per BOTC as representatives. We define BOTCs formally as follows:

Definition (Business Object Testing Class – BOTC): *A business object testing class* $\mathcal{G}_i$ *is a function returning a set of semantically similar data items. It is applicable to customer data before and after the upgrade –* $\mathcal{G}_i(\mathcal{D}^b_{cus}) \subseteq \mathcal{D}^b_{cus}$ *respectively* $\mathcal{G}_i(\mathcal{D}^u_{cus}) \subseteq \mathcal{D}^u_{cus}$. $\tilde{\mathcal{G}}$ *denotes the set of all BOTCs.*

We want to have all (relevant) data items after an upgrade, and no (relevant) data items are allowed to emerge during the upgrade (*completeness correctness*). The BOTCs enable us to check this, because the set of all BOTC data items is the set of relevant data items before respectively after the upgrade.

A formal definition of completeness correctness requires, first, that we can express that two data items before and after the upgrade represent the same real world object. We name this concept *data correspondence*. Based on it, we define completeness correctness formally.

Definition (Data Correspondence): *Let two data items* $d^b_i \in \mathcal{D}^b_{cus}$ *and* $d^u_i \in \mathcal{D}^u_{cus}$ *represent the same data before and after the upgrade. Then, they correspond:* $d^b_i \mapsto d^u_j$.

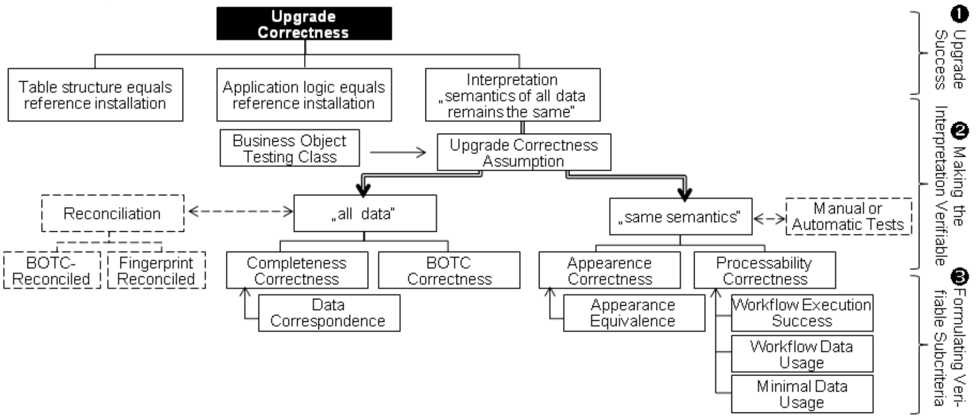
Upgrade Correctness
Table structure equals reference installation
Application logic equals reference installation
Interpretation „semantics of all data remains the same"
Business Object Testing Class
Upgrade Correctness Assumption
Reconciliation
„all data"
„same semantics"
Manual or Automatic Tests
BOTC-Reconciled
Fingerprint Reconciled
Completeness Correctness
BOTC Correctness
Appearence Correctness
Processability Correctness
Data Correspondence
Appearance Equivalence
Workflow Execution Success
Workflow Data Usage
Minimal Data Usage
❶ Upgrade Success
❷ Making the Interpretation Verifiable
❸ Formulating Verifiable Subcriteria

Figure 5: Correctness and Testing Concepts

Definition (Completeness Correctness): *An upgrade UPG is completeness correct – CoCo(UPG) – if and only if exactly the data items belonging to the BOTCs before the upgrade are the ones belonging to the BOTCs afterward and vice versa:*

$$CoCo(UPG) \Leftrightarrow$$
$$\forall d^b \in \bigcup_i \mathcal{G}_i\,(D^b){:}\ \exists d^u \in \bigcup_i \mathcal{G}_i\,(D^u){:}\ d^b \mapsto d^u$$
$$\wedge\ \forall d^u \in \bigcup_i \mathcal{G}_i\,(D^u){:}\ \exists d^b \in \bigcup_i \mathcal{G}_i\,(D^b){:}\ d^b \mapsto d^u$$

Our tests rely on typical data items for each BOTC. Thus, data items must not change the BOTCs during the upgrade. We name this property *BOTC correctness*.[1]

Definition (BOTC Correctness): *An upgrade UPG is BOTC Correct – BCo(UPG) – if and only if the data items belong to the same BOTCs before and after the upgrade:*

$$BCo(UPG) \Leftrightarrow\ \ \forall \mathcal{G}_i \in \tilde{\mathcal{G}}\ \forall d^b \in \mathcal{G}_i(D^b) \exists d^u \in \mathcal{G}_i(D^u){:}\ d^b \mapsto d^u$$
$$\wedge\ \forall \mathcal{G}_i \in \tilde{\mathcal{G}}\ \forall d^u \in \mathcal{G}_i(D^u) \exists d^b \in \mathcal{G}_i(D^b){:}\ d^b \mapsto d^u$$

If an upgrade is completeness correct and BOTC correct, all data items of all BOTCs have been upgraded. Now we can address the two aspects of "same semantics" (Figure 5): the appearance and the processability of the data. The appearance of the data means that the GUI presents semantically identical data items before and after the upgrade. Here, the typical data items of each BOTC come into play. Testers compare them on the GUI before and after the upgrade. The term *appearance equivalence* covers the knowledge about data items before and after the upgrade. We might know that the data items are semantically equivalent. We might know they are not. Finally, we might have no knowledge.

Definition (Appearance Equivalence): *Let d^b and d^u be data correspondent data items ($d^b \mapsto d^u$) before and after an upgrade. Our knowledge about their appearance after the upgrade can be either:*

- *We know both are semantically equivalent: $d^b \overset{\checkmark}{\Rightarrow} d^u$*
- *We know they are not equivalent: $d^b \overset{\neq}{\Rightarrow} d^u$*
- *We have no information about the data items: $d^b \overset{?}{\Rightarrow} d^u$*

We assume an upgrade to be *appearance correct* if we know at least one data item per BOTC that it is semantically equivalent and if we do not know any non-equivalent data item.

Definition (Appearance Correctness): *An upgrade UPG is appearance correct – AppCo(UPG) – if we know that there is at least one appearance correct data item per BOTC and none which is known not to be.*

$$CACo(UPG) \Leftrightarrow$$
$$\forall \mathcal{G}_i \in \tilde{\mathcal{G}}\ \exists \langle d^b, d^u \rangle \in \{\langle d', d'' \rangle | d' \in \mathcal{G}_i(D^b) \wedge d'' \in \mathcal{G}_i(D^u) \wedge d' \mapsto d''\}{:}$$
$$d^b \overset{\checkmark}{\Rightarrow} d^u \qquad \wedge$$
$$\forall \mathcal{G}_i \in \tilde{\mathcal{G}}\ \nexists \langle d^b, d^u \rangle \in \{\langle d', d'' \rangle | d' \in \mathcal{G}_i(D^b) \wedge d'' \in \mathcal{G}_i(D^u) \wedge d' \mapsto d''\}{:}$$
$$d^b \overset{\neq}{\Rightarrow} d^u$$

Appearance correctness is important, but does not detect all data semantics problems. An example: A new attribute is added to a table. If the attribute does not have a suitable value, workflows might crash. If, e.g., there is no NOT NULL schema constraint, the database cannot enforce this property. This is typical for complex constraints spanning various tables. We can detect such problems by executing workflows which thereby use and "test" data items. Formally:

Definition (Workflow Execution): *A workflow execution is a triple $e_i = \langle w_i, \overline{D}_i, s_i \rangle$. $w_i \in W^u$ is the executed workflow. $\overline{D}_i \subseteq D^u$ are the data items used in the execution. $s_i \in \{success, failure\}$ states the execution result.*

Generic rules would help stating which workflows shall be tested with which data items. This is not realistic due to the complexity of commercial information systems. Thus, we demand only that at least one data item per BOTC is used in a workflow execution. We name this the *minimal processability correctness* heuristic.

Definition (Minimal Processability Correctness): *An upgrade UPG is minimal processability correct – MPCo(UPG) – if the set of executed workflows Ê (i) covers data items of all BOTCs and none of the workflow executions fails.*

$$MPCo(UPG) \Leftrightarrow \forall \mathcal{G}_i \in \tilde{\mathcal{G}}{:}\ \exists \langle w^u, \overline{D}^u, s^u \rangle \in \hat{E}{:}\ \exists d \in \overline{D}^u{:}\ d \in \mathcal{G}_i(D^u)$$
$$\wedge\ \forall \langle w', D', s' \rangle \in \hat{E}{:}\ \ s' = 'success'$$

If an upgrade fulfills our four criteria – CoCo, BCo, AppCo, and MPCo – we know that, first, all relevant data items have been upgraded. We know, second, that the semantics of typical data items are unchanged. Third, we are not aware of any incorrect data item. So we *assume* the interpretation of the customer data to be unchanged. This is the upgrade correctness assumption we stated above. Formally:

Definition (Upgrade Correctness Assumption): *An upgrade is upgrade correct, if and only if it is completeness correct, BOTC correct, appearance correct and processability correct:*

$$UPGCo(UPG) \Leftrightarrow CoCo(UPG) \wedge BCo(UPG)$$
$$\wedge\ CACo(UPG) \wedge MPCo(UPG)$$

V. Efficient Testing in Practise

A. Completeness and BOTC Correctness

Testing completeness correctness and BOTC correctness is like searching for a needle in a haystack, or, more precisely, ensuring there is no needle in the haystack. Our information system can have hundreds of tables. It can have millions of rows. Still, we must neither lose one single (relevant) data item, "create" a new one, nor shall we assign data items to a different BOTC. Obviously, these tests must be done automatically. Thus, we rely on a concept known from data migration named *reconciliation* [13]. We discuss two basic reconciliation strategies: BOTC and fingerprint reconciliation.

A **BOTC reconciliation** counts the data items of each BOTC before and after the upgrade. Table 2 is an example. The rows represent various BOTCs of a bank: customers, accounts, and loans. The columns contain the number of data items before (Table 2, column "Before") and after the upgrade (column "Upgraded"). If they are the same for all BOTCs, the information system is **BOTC reconciled**. It indicates completeness and BOTC correctness.

[1] Obviously, BOTC correctness ensures also completeness correctness.

TABLE 2: BOTC RECONCILIATION EXAMPLE

BOTC	Before	Upgraded	OK?
Customers	157'030	157'030	✓
Accounts	378'051	378'051	✓
Loans	39'000	37'407	×

The underlying assumption is that "losing" or "creating" data items during the upgrade changes the number of data items in a BOTC. Based on our practical experience, this approach is highly reliable. However, it has one drawback. It detects *that* there is a mistake. It cannot explain *what* is wrong. A fingerprint reconciliation overcomes this drawback.

A **fingerprint reconciliation** checks which data items before and after the upgrade (do not) have a counterpart. It matches them based on IDs. Many data items have such an ID (e.g., a unique account ID for bank accounts). If not, we construct a synthetic ID. A fingerprint reconciliation iterates over all BOTCs (see Table 3): first all accounts, then the loans, etc. Next, it matches the IDs for each BOTC before and after an upgrade. There are matching accounts with IDs 278512 and 278512 before and after the upgrade. There are no matching IDs for the two loans. ID ZH5667 exists only before the upgrade, ZH5668 only afterwards. A fingerprint reconciliation with three columns BOTC, id_b, and id_u is sufficient for checking completeness correctness and BOTC correctness. One can even include semantic information by using fingerprints (f_b and f_u). Fingerprints might include the account balance (32000 and -2312) and the owner's ID (45522 and 2334). So we would notice that account 278513 has still the correct balance (-2312), but that the owner ID changed from 2334 to 3836.

We name an information system **fingerprint reconciled** if for all rows of all (relevant) tables the following condition holds: first, there are matching IDs and, second, if there are fingerprints, they are equal.

B. Appearance and Processability Correctness

Testing appearance and processability correctness addresses the semantics of the upgraded data. It is (in principle) a manual task. The costs rise linearly with the number of BOTCs (*not* data items) and retests. This calls for an efficient testing process. There are two easy measures: availability of "good" test data and automatic tests.

Availability of good test data does not address the linear nature of the costs. It makes the retests itself more efficient by reducing the preparation efforts. Testers need characteristic data items for each BOTC. Only then, they can compare the data before and after the upgrade (appearance correctness) or execute workflows with data (processability correctness). If such data exists and is documented well, even new testers can start testing immediately. How one derives the test data (e.g., live test data or manually defined data) depends on many factors (see [14] for more details).

TABLE 3: FINGERPRINT RECONCILIATION EXAMPLE

BOTC	id_b	id_u	f_b	f_u	OK?
Account	278512	278512	32000|45522	32000|45522	✓
Account	278513	278513	-2312|2334	-2312|3836	×
Loan	ZH5667	NULL	230:120	NULL	×
Loan	NULL	ZH5668	NULL	50:140	×

A second way to improve efficiency is test automation. It addresses the linear cost nature. Many testing suites on the market automate regression tests (e.g., Quick Test Professional [15]). However, one has to consider the semantics related risks in case of test automation. First, a GUI might look the same after an upgrade, but might have a new meaning. The GUI title (and semantics) could change, e.g., from "Balance Sheet Previous Year" to "Balance Sheet Current Year." Second, the unit of measurement can change (e.g., from £ to $,) or one could write "5 Mio" instead of "5'000'000". Third, meaning and name of a field can change (e.g., "liabilities" instead of "profit"). Finally, there is the non automation-specific risk. The link between an attribute and row in the database to a specific GUI field might change. One must be aware of the risk, but one should not reject automation per se as completely impossible.

VI. SUMMARY

First, we described in this paper our three standard processes for upgrades: install & copy, rejuvenation/delta only, and rejuvenation/verified. Second, our upgrade success criteria elaborate when it is safe to assume an upgrade to be correct. Crucial is dividing the correctness problem into two parts: first, ensuring that "all data" is available after an upgrade, and, second, checking typical data items for whether their semantics are unchanged. Third, we presented implementation strategies for making our theoretical insights useful in practice, which we derived from our various commercial projects with banks. Thus, this paper overcomes an important gap in information system and software maintenance research: how to develop and test information system upgrades systematically.

ACKNOWLEDGEMENTS. The author would like to thank Tim Weingärtner for the valuable discussions.

REFERENCES

[1] R. Lämmel, "Coupled Software Transformations (Extended Abstract)," First Int. Workshop on Software Evolution Transformations, 2004

[2] B. Agnew, et al., "Planning for Change: A Reconfiguration Language for Distributed Systems," Distributed Systems Engin., Vol. 1(5), 1994

[3] H. Malik, A. Hassan, "Supporting Software Evolution Using Adaptive Change Propagation Heuristics," ICSM'08, Beijing, China

[4] S. Jansen, et al., "A Process Model and Typology for Software Product Updaters," CSMR'05, March 21–23, 2005, Manchester, UK

[5] J. Rellermeyer, et al., "Consistently Applying Updates to Compositions of Distributed OSGi Modules," HotSWUp'08, Nashville, TN. 2008

[6] S. Ajmani, et al., "Modular Software Upgrades for Distributed Systems," ECOOP'06, July 3–7, 2006, Nantes, France

[7] R. Oreizy, et al., "Architecture-Based Runtime Software Evolution," ICSE'98, April 19–25, 1998, Kyoto, Japan

[8] C. Curino, et al., "Automating Database Schema Evolution in Information System Upgrades," HotSWUp'09, Orlando, FL, 2009

[9] C. Yu, L. Popa, "Semantic Adaptation of Schema Mappings when Schemas Evolve," VLDB, Aug. 30–Sept. 2, 2005, Trondheim, Norway

[10] R. Beatty, C. Williams, "ERP II: Best Practices For Successfully Implementing an ERP Upgrade," CACM, Vol. 49(3):105–109 (2006)

[11] G. Biermann, et al., "Formalizing Dynamic Software Updating," Workshop on Unanticipated Software Evolution, 2003, Warsaw, Poland

[12] K. Haller, "Information System Maintenance Costs: The 'In-between' Challenge," Workshop Software-Reengineering, Bad-Honnef, 2010

[13] K. Haller, "Towards the Industrialization of Data Migration," CAiSE'09, June 8–12, 2009, Amsterdam, The Netherlands

[14] K. Haller, "The Test Data Challenge for Database-Driven Applications," DBTest 2010, June 7, 2010, Indianapolis, IN

[15] HP Quick Test Professional, http://www.hp.com