Modeling Data for Business Processes

Yutian Sun\textsuperscript{1}, Jianwen Su\textsuperscript{1}, Budan Wu\textsuperscript{†}, and Jian Yang\textsuperscript{‡2}
\textsuperscript{1} Department of Computer Science, UC Santa Barbara, USA
\textsuperscript{†} Institute of Network & Technology, Beijing Univ. of Posts & Telecom., China
\textsuperscript{‡} Department of Computing, Macquarie University, Australia

Abstract—An important omission in current development practice for business process (or workflow) management systems is modeling of data & access for a business process, including relationship of the process data and the persistent data in the underlying enterprise database(s). This paper develops and studies a new approach to modeling data for business processes: representing data used by a process as a hierarchically structured business entity with (i) keys, local keys, and update constraints, and (ii) a set of data mapping rules defining exact correspondence between entity data values and values in the enterprise database. This paper makes the following technical contributions: (1) A data mapping language is formulated based on path expressions, and shown to coincide with a subclass of the schema mapping language Clio. (2) Two new notions are formulated: Updatability allows each update on a business entity (or database) to be translated to updates on the database (or resp. business entity), a fundamental requirement for process implementation. Isolation reflects that updates by one process execution do not alter data used by another running process. The property provides an important clue in process design. (3) Decision algorithms for updatability and isolation are presented, and they can be easily adapted for data mappings expressed in the subclass of Clio.

I. INTRODUCTION

Two key components in modern enterprise systems are data management and business process management (BPM). Data refer to the persistent data managed by DBMS. Business processes (BPs) prescribe how business operations should be conducted. BPs consume, manipulate, and generate data; their interoperation is often accomplished through sharing data access. Current data and BP modeling approaches (e.g., ER [7], BPMN [5]) leave associations of persistent data in databases and data in BPs to the implementation level with little abstraction. Implementing business logic involves data access from and to database and often demands high development efforts.

Recognizing the importance of integrating data with process, the BPM community is embracing a shift from traditional activity-centric BP models (e.g. [25]) to data-centric modeling. Artifact models [19], [4] lead this trend by using an information model for data in a BP and a lifecycle model to capture how the business data evolve through business operations in the BP. In object-centric models [21], [16], process logic was modeled as object behaviors and object coordination. However, the modeling and design of the connection between databases and BPs is still missing. The challenge in system support for “linking” database and BP has two aspects: 1. A formal approach that models the business process behavior and its associated data, captures its running status relevant to database updates, and maintains its connection with the database at all time. 2. Modeling and tool support is needed for automation and to ensure that every BP runs on its own data and maintains data consistency with the database.

To address the challenges, we present a formal model for connecting artifact BPs and databases. It consists of three parts: (1) a database schema that specifies the data structure in the persistent store, (2) an artifact system that specifies hierarchical structures for BP data as business entities along with access dependencies and constraints, and (3) entity-database mappings to manage data accesses and updates between database and BPs. The approach allows us to achieve:

- Model and manage BPs and database separately: a BP only accesses data in its entity, the database is connected the BP through the entity rather than directly (without abstraction).
- BP data and the database are conceptually separated but remain “connected” (via mapping rules) and consistent (through the updatability property).
- Isolation property reflects the situation that data updates in one BP will not affect another BP execution in relation to the propagated database change.

This paper makes the following technical contributions: (1) based on formal models for database and artifact (BP) design, a language is developed for specifying mapping rules between database and artifacts. It is shown that the language coincides with a subclass of Clio [11]. (2) Two new notions are formulated. “Updatability” allows each update on a business entity (or database) to be translated to updates on the database (or resp. business entity), a fundamental requirement for (business) process implementation. “Isolation” requires that updates by one process execution does not alter data used by another running process. The property provides an important clue in process design. (3) Decision algorithms for updatability and isolation are presented, which can be adapted for data mappings in the subclass of Clio.

This paper continues the study on artifact BPs (e.g., [4], [22]) and is an initial effort in formal modeling of relationships between enterprise database and BP data so that the BP data can be separately managed and yet integrated with the database. Our work leads to a new, promising framework for an integrated approach in modeling BPs and persistent data in enterprise systems.

The remainder of the paper is organized as follows. §II motivates the framework and technical problems using a real example. §III provides the essential concepts for the technical discussions. §IV introduces the mapping language, and establishes the equivalence of the language and a subclass of Clio. §V formulates and studies the updatability concept.

\textsuperscript{1}Supported in part by a grant from Bosch.
\textsuperscript{2}Supported in part by a grant from ARC LP120200231.
Application Review
Repair Application
r(Customer Name)
w(Customer Name)
... 
Parts
Repairperson Assignment
w(Service ID)
w(Repairs-person Name)
... 
Document Archive
On-site Repair
w(Material ID)
w(Material)
... 
Post-repair Visit
w(Service ID)
... . . . . . . . .

Fig. 1: A Business Process for RBPS

§VI focuses on the isolation property. Related work and conclusions are provided in §VII and §VIII, resp.

II. Motivations

In this section we describe an application development that demands a framework to model data accessed by a BP and automate database accesses by the BP through specified "mappings" between the BP data and the underlying database. Since the database is modified by BP executions indirectly through the data mappings, two technical problems arise: (1) "updatability", the ability to translate data modifications by BPs to database updates, and (2) "isolation", a property that data modifications by one process execution do not implicitly change the data used by another.

Kingfore Corporation (KFC, www.kingfore.net) in Beijing was developing a repair management system (RMS) to manage their business of heating equipment repairs. Fig. 1 shows the (simplified) primary process with six activities (shown as boxes); inside each box are attributes to be read or written by the activity. Actual values of all attributes are stored in a single database.

The process is initiated by a repair request from a customer via the RMS web front; a KFC operator is notified and approves the application if an on-site repair is needed. The application is then sent to the corresponding heating center manager for the customer's residential location. The manager reviews the request, and assigns one or more service-persons, a.k.a. repairs-persons for the case. After the repair is completed (with one or more site visits) a representative from KFC visits the customer and closes the case. However, if the service-persons are unable to fix the problem or the post-repair visit receives an unsatisfactory response, the request is sent back to the manager and new service-persons will be assigned.

Once business entities are modeled, associations of business entities and the database can be defined as "data mappings". With such mappings between business entities and the database, the process designer only needs to focus on business entities in process modeling/design, while ignoring the database. (In general, the database may contain much more data than what is needed by a single process.)

Fig. 3 shows a part of the database schema used by KFC, where keys are underlined, foreign keys italicized with references as arrows. The database includes six relation schemas: tUser (information of customer and staff including repairs-persons) with key (tLastName, tFirstName), tRepair (customer repair requests) with key tRepairID and a foreign key (tCustomerLN, tCustomerFN) to tUser (the requesting customer), tServiceInfo (individual on-site repair services performed by repairs-persons), tRepairs-person (the assigned repairs-persons in a single service), tReview (service reviews) with two keys tReviewID and tServiceID_R, and tMaterialInfo (replacement parts used in a service) whose foreign key tServiceID_MI is also in a key.

Consider as an example where the aID value of each instance of the business entity (Fig. 2) for the repair process is mapped to the tRepairID value of some tuple in the table tRepair (Fig. 3). Then, the aCustomerLN and aCustomerFN values in the business entity instance with aID = 101 should correspond to tCustomerLN and tCustomerFN values (resp.) of the tuple with matching tRepairID (i.e., = 101). This can be specified using a path expression for aCustomerLN: “aCustomerLN=aCustomerLN”., where the first half of the expression navigates in the business entity (Fig. 2) and the second half in the database (Fig. 3). More specifically, from the attribute-value pair aCustomerLN in a business entity instance, we are able to uniquely locate the attribute-value pair aCustomer, and consequently uniquely locate the attribute-value pair aID. With the value of aID, we can match the value of tRepairID in table tRepair and then find the value of tCustomerLN (in the corresponding tuple). The above expression means that the given value of aCustomerLN should be identical to the value of tCustomerLN. Furthermore, the values of aCustomerLN and aCustomerFN in the business entity instance with aID = 101 can be fetched by an SQL query:

```
SELECT tCustomerLN, tCustomerFN
FROM tRepair
WHERE tRepairID=101
```

This example hints that one can generate SQL expressions for database accesses from the data mappings. A further advantage of the mappings is that modification of a process (or database) can be made locally on the process (resp. database) first and then the mappings are adjusted. The current practice is to consider both at the same time, a more complex task.
During the RMS development, a problem encountered was that some updates on a business entity could not be translated into database updates. Consider the following mapping expressed using Clio [11] from the database (Fig. 3) as the source to the business entity (Fig. 2) as the target.

\[ \forall i, i_m, n_1, n_2, n_f, m \ t \text{ServiceInfo}(i_1, 101, t), \ t\text{Repairperson}(i_1, n_i, n_f), \text{MaterialInfo}(i, i_m, m) \rightarrow \exists R \ a\text{Repairperson}(n_1, n_2, R), R(p, m) \]  

Rule (\(\ast\)) creates an \(a\text{Repair}\) instance. Each entry in \(a\text{Repair}\) records for each service the replacement parts used by a service-person (using a join of \(t\text{MaterialInfo}\) and \(t\text{Repairperson}\) on the service ID). Suppose there are two tuples \((a_1, b, c_1)\) and \((a_2, b, c_2)\) in \(t\text{MaterialInfo}\), and two tuples \((b, d_1, e_1)\) and \((b, d_2, e_2)\) in \(t\text{Repairperson}\). Then \(a\text{RepairPerson}\) should have four entries \((d_1, e_1, (p_1, c_1)), (d_1, e_1, (p_2, c_2)), (d_2, e_2, (p_3, c_1)), (d_2, e_2, (p_4, c_2))\), where \(p_1, p_2, p_3, p_4\) are some phone number values. If the process execution is to delete the entry \((d_2, e_2, (p_3, c_1))\) in \(a\text{RepairPerson}\), it is not possible to update the database so that the mapping rule (\(\ast\)) would give the updated \(a\text{RepairPerson}\). The reason that rule (\(\ast\)) is not “updatable” is due to poor design, which makes a cross product of tables \(t\text{MaterialInfo}\) and \(t\text{RepairPerson}\), and then stores the result in the same tuple \(a\text{RepairPerson}\). A solution to this problem is to move the attribute \(a\text{Misc}\) (a dashed box in Fig. 2) that stores the information of replacement parts to another attribute \(a\text{Part}\) under the tuple \(a\text{ReplacementParts}\). For the remainder of this paper, we consider the attribute \(a\text{Misc}\) removed.

Another situation that can lead to an inconsistent update is that two attributes in the same business entity (instance) are mapped to the value of the same tuple in the database.

**Example 2.1:** Consider an instance of a business entity in Fig. 2, where both attributes \(a\text{Repair_Addr}\) and \(a\text{Cust_Addr}\) are mapped to the attribute \(t\text{Address}\) in the same tuple in table \(t\text{User}\) (Fig. 3). When the value of \(a\text{Repair_Addr}\) is updated to the one that is different from the value of \(a\text{Cust_Addr}\), the database is unable to capture this change. The problem lies in the redundancy of information stored in the business entity. A quick fix is to remove a redundant attribute \(a\text{Repair_Addr}\).

Similarly, for the remainder of this paper, we also consider attribute \(a\text{Repair_Addr}\) removed.

Since the data in a process are mapped into the database, the above discussion suggests a critical “updatability” property that every business entity modification by a process should always be translated into database updates. In this paper, we argue that each specified mapping should be “updatable”.

Another interesting property is independence between two process executions, called “isolation”, i.e., the two process instances will not update the same attribute of the same tuple in the database. Consider the entity in Fig. 2. Suppose there are two running repair process instances requested by the same customer; if \(a\text{Cust_Addr}\) is updated by one of the instances, the other repair process instance for the same customer will “see” the changed address even though it makes no updates on the address. In this situation, the artifact (in Fig. 2) is not “isolated” (with itself). For this example, it is desirable to have the address change made by one process be immediately visible by the other running instance. However, Such implicit changes are not always helpful.

For example, an address change in the repair process of a customer will alter the address in an ongoing “Customer Profile Update” instance by the same customer. This failure to isolate is counter-intuitive and not desirable. To avoid this situation, one can restrict attribute \(a\text{Cust_Addr}\) in Fig. 2 to be “read-only” for the repair process.

Isolation property is important to BP designers. Unlike up-datability, we do not require artifacts to be always “isolated”.

### III. DATABASE AND ARTIFACT MODELS

This section provides key notions of the relational data model [8] and the information model of an artifact-centric process model [19] for process design (our formalism is closer to the one defined in [4]).

#### Relational databases

The data model includes “keys” and “foreign keys” with cardinality bounds that resemble the entity-relationship model [7] with cardinalities.
For the technical development, we assume a totally ordered set of names that are used as attribute names, relation names, and names in the artifact model presented later in the section. Every (finite) set of names is enumerated according to this total order. Let \( \mathbb{N} \) be the set of natural numbers and \( \mathbb{N}^+ = \mathbb{N} - \{0\} \).

**Definition:** A relation schema is a tuple \((R, A, K)\), where \(R\) is a name for the relation schema, \(A\) a finite set of names for (primitive) attributes and \(R \notin A\), and \(K \subseteq 2^A\) a set of keys over \(R\) such that no single key is properly contained in another. Each attribute in a key in \(K\) is called prime.

We also denote a relation schema \((R, A, K)\) as \(R(A, K)\) or simply \(R\) when it is clear from the context, \(A\) as \(\text{Attr}(R)\), and \(K\) as \(\text{Keys}(R)\). We omit types of attributes (that can be easily added). Thus we assume the existence of a universal domain \(\text{Dom}\) that also contains a special symbol ‘\(\lambda\)’ to denote the situation when an attribute value is undefined.

**Definition:** Given a relation schema \(R(A, K)\), a tuple (of \(R\)) is a total mapping \(t: A \rightarrow \text{Dom}\) such that \(t(a) \neq \lambda\) if \(a\) is prime. A (relation) instance (of \(R\)) is a finite set of tuples such that their key values are pairwise distinct (unique).

A database schema consists of a set of relation schemata, and a set of “foreign keys”, each having a cardinality bound. Similar to cardinality constraints in the ER model \([7]\), a bound limits the number of occurrences of a foreign key value.

**Definition:** A database schema is a triple \((\mathbb{R}, F, \lambda)\), where

- \(\mathbb{R}\) is a set of relation schemata with distinct names and pairwise disjoint attribute sets. Let \(\text{Keys}(\mathbb{R}) = \bigcup\{\text{Keys}(R) \mid R \in \mathbb{R}\}\) be the set of all keys in \(\mathbb{R}\).
- \(F \subseteq (\bigcup_{R \in \mathbb{R}} 2^{\text{Attr}(R)}) \times \text{Keys}(\mathbb{R})\) is a set of foreign keys such that (1) for each pair \((S, \kappa)\) in \(F\), \(|S| = |\kappa|\), \(S\) is not a proper superset of a key in \(\text{Keys}(\mathbb{R})\), and if \(S\) is a subset of attributes in a relation schema \(R\) and \(\kappa \in \text{Keys}(R')\), then \(R' \neq R\); and (2) the graph \((\bigcup_{R \in \mathbb{R}} 2^{\text{Attr}(R)}, F)\) is acyclic.
- \(A: F \rightarrow \{1, ?, +, *\}\) is a total mapping assigning a cardinality bound to each foreign key, where for each foreign key \(f, \lambda(f)\) limits the number of occurrences of each \(f\)-value with 1 stands for exactly one, \(?\) for at most once, \(+\) for at least once, and \(*\) for unrestricted.

Fig. 3 shows (part of) the database schema of the maintenance company discussed in Section II.

**Definition:** Given a database schema \(S = (\mathbb{R}, F, \lambda)\), a database (instance) of \(S\) is a total mapping \(d\) from \(\mathbb{R}\) to instances of \(\mathbb{R}\) such that for each \(R \in \mathbb{R}\), \(d(R)\) is an instance of \(R\) and \(d\) satisfies all foreign key constraints in \(F\) and the cardinality bounds in \(\lambda\). Let \(\text{inst}(S)\) be the set of all databases of \(S\).

The definition omits conditions for satisfaction of foreign keys and cardinality bounds: the former can be found in, e.g. \([2]\), and the latter means that each value for a key must occur in the referencing relation for the specified number of times.

In this paper, three types of database updates are considered: insertion, deletion, and modification. Given a database \(d\), an insertion, deletion, and modification operation on \(d\) adds a new tuple to \(d\), removes a tuple from \(d\), and, resp., changes the value of a non-prime attribute in a given tuple in \(d\). We only consider updates whose the result database satisfies all key and foreign key constraints. For a database schema \(S\), let \(\Delta_S\) be the set of all possible updates on \(\text{inst}(S)\). Given a database \(d \in \text{inst}(S)\), for each \(\delta \in \Delta_S\), \(\delta(d)\) denotes the resulting database after applying \(\delta\) on \(d\).

**Artifacts and business entities**

Artifact-centric models \([19], [15], [27]\) specify a process with a data model and a lifecycle. In the remainder of the section, we formulate the key concepts of the artifact model used in technical discussions. The following formalizes hierarchical structures for business entities.

**Definition:** The family of (complex) attributes is recursively defined as follows.

- each primitive attribute is an attribute,
- “\(a: (a_1, ..., a_n)\)” is a tuple attribute and “\(a: \{a_1, ..., a_n\}\)” is a set attribute, if \(n \geq 1\), \(a_i\)’s are attributes, and \(a\) is a name not occurring in any of the \(a_i\)’s.

Let \(\text{Attr}(a)\) denote the set of all attributes used in \(a\) (including the name for \(a\)), and \(\text{Prim}(a)\) the set of primitive attributes in \(a\). A value of an attribute \(a\) is defined as follows.

- Each element in \(\text{Dom}\) is a value of a primitive attribute \(a\),
- \(\{a_1: v_1, ..., a_n: v_n\}\) is a value of a tuple attribute \(a: (a_1, ..., a_n)\) if each \(v_i\) is a value of \(a_i\),
- Each finite (possibly empty) set \(\{a_1: v_1, ..., a_n: v_n\}\) is a value of a set attribute \(a: \{a_1, ..., a_n\}\) if \(k \in \mathbb{N}\) and for each \(i \in [1..k]\) and each \(j \in [1..n]\), \(v_{i,j}\) is a value of \(a_j\).

Given a tuple attribute “\(a: (a_1, ..., a_n)\)” or a set attribute “\(a: \{a_1, ..., a_n\}\)” for each \(i, j \in [1..n]\), \(a_i\) is a child (attribute) of \(a\) and a sibling (attribute) of \(a_j\), \(a_i\) is the parent (attribute) of \(a_j\). Let \(a\) be an attribute. A set \(\kappa\) of attributes in \(\text{Attr}(a)\) is a key (in \(a\)) if attributes in \(\kappa\) are pairwise siblings of each other. In our model, key values must be unique among all artifact “enactments” in a “snapshot”. A local key (in \(a\)) is a pair \((\kappa, b)\) where \(\kappa\) is a key in \(a\) and \(b\) an attribute in \(a\) and an ancestor of every attribute in \(\kappa\). Intuitively, \(b\) provides the “context” within which a local key value is unique.

Given a set \(V\) of values for a complex attribute \(a\), a key \(\kappa\) in \(a\), a local key \((\kappa, b)\) in \(a\), \(V\) satisfies the key \(\kappa\) if there are no undefined values for attributes in \(\kappa\) and each value of \(\kappa\) attributes occurs at most once (i.e., is unique in \(V\)). \(V\) satisfies the local key \((\kappa, b)\) if there are no undefined values for attributes in \(\kappa\) and each value of \(\kappa\) attributes occurs at most once within each value of \(b\). Let \(\text{Value}(a)\) denotes the set of all values of \(a\) in \(\text{inst}(S)\).

In order to define the notions of artifacts, business entities, and mapping rules, we introduce “functional paths” (for retrieving an attribute from another attribute).

Given a complex attribute \(a\), a functional (or function) path \(p\) from \(a_1\) to \(a_n\) is an expression of form “\(a_1, a_2, \ldots, a_n\)”, where \(n \geq 0\), \(a_i \in \text{Attr}(a)\) for each \(i \in [1..n]\), and for each \(i \in [1..(n-1)]\), \(a_{i+1}\) is a sibling, parent of \(a_i\), or a child of \(a_i\) if \(a_i\) is not a set
attribute. Also, \( a_1 \) and \( a_n \) are the head and tail of \( p \), resp. Note that if \( a_1, \cdots, a_n \) is a fun-path, so is \( a_{i_1}, \cdots, a_{i_j} \) for all \( 1 \leq i \leq j \leq n \).

Intuitively, a functional path from attribute \( a \) to \( b \) denotes that given an attribute-value pair of \( a \), a unique value of \( b \) can be determined. In general, not every pair of attributes have a functional path as they may go through set attributes, where multiple values of \( b \) can be obtained given \( a \).

A “business entity” is a complex attribute with (local) keys and “access dependencies”. The latter specifies a partial order on all primitive attributes in the complex attribute to denote that an attribute can be read/written after other attributes have been written. Although access dependencies could be divided into “write-write” and “write-read” dependencies, we combine them since they do not affect the technical results.

**Definition:** A business entity (or bentity for short) is a tuple \( (\Omega, a, K, L, \text{dep}) \), where

- \( \Omega \) is a (unique) name,
- \( a \) is a (complex) attribute with name \( \Omega \) and \( a \) contains the primitive attribute “ID” as its child,
- \( K \) is a set of keys in \( a \) and \( L \) a set of local keys in \( a \) such that (1) \([\text{ID}] \in K \), (2) each (local) key in \((K \cup L)\) other than \([\text{ID}]\) has a set attribute as its parent, and (3) each set attribute in \( a \) has exactly one element in \( K \cup L \) as its child(ren), and
- \( \text{dep} \subseteq \text{Pmt}(a) \times \text{Pmt}(a) \) is a set of access dependencies such that the graph induced by \( \text{dep} \) is a directed acyclic graph rooted at \( \text{ID} \) and for each edge \((u, v)\) in this graph, there exists a “functional path” from \( v \) to \( u \).

When it is clear from the context, we may conveniently denote \( (\Omega, a, K, L, \text{dep}) \) as \( (\Omega(a, K, L, \text{dep})) \), or simply \( \Omega \).

**Example 3.1:** Fig. 2 shows the bentity of the business process (artifact) discussed in Section II. Each repair has an ID (renamed to \( aID \) to avoid confusion), repair information (\( a\text{Repair}_\text{Info} \)), and several services (\( a\text{Service}_\text{Info} \)). Each service may require multiple replacement parts (\( a\text{Replacement}_\text{Parts} \)), 0 or 1 review (\( a\text{Review}_\text{Info} \)). Set attributes (e.g., \( a\text{Service}_\text{Info} \) and \( a\text{Replacement}_\text{Parts} \)) are attached with a circled-star. There are three keys, \( a\text{ID} \), \( a\text{RP}_\text{Last}_\text{Name} \), \( a\text{RP}_\text{First}_\text{Name} \) and \( a\text{Service}_\text{ID} \) (indicated by \( \text{UNIQUE} \)) and a local key \( a\text{Part}_\text{ID} \) within the context of \( a\text{Replacement}_\text{Parts} \) (UNIQUE IN). Primitive attributes include \( a\text{ID} \), \( a\text{Repair}_\text{Info} \), \( a\text{Cust}_\text{Name} \), \( a\text{Service}_\text{ID} \), etc. An access dependency example could be that before writing \( a\text{Cust}_\text{Name} \), \( a\text{ID} \) should be written, or writing \( a\text{Cust}_\text{Name} \) precedes reading \( a\text{Cust}_\text{Addr} \). An example fun-path can be “\( a\text{RP}_\text{Last}_\text{Name}\text{.aRepair}_\text{person}.a\text{Service}_\text{Info}.a\text{ID} \)” which means that given the last name of a repair person in a bentity instance, their is only one \( a\text{ID} \) we can obtain. On the other hand, given an \( a\text{ID} \), there could possibly have multiple last names (of \( a\text{Repair}_\text{person} \)).

**Definition:** An enactment of a bentity \( \Omega(a, K, L, \text{dep}) \) is a value of \( a \) that satisfies all local keys in \( L \). An instance of \( \Omega \) is a finite set of enactments of \( \Omega \) that satisfies each key in \( K \). We denote by \( \text{Ent}(\Omega) \) (or \( \text{inst}(\Omega) \)) the set of all enactments (resp. instances) of \( \Omega \).

**Definition:** A (bentity) system \( \mathcal{A} \) is a finite set of bentities whose attribute sets are pairwise disjoint. A snapshot of \( \mathcal{A} \) is a total mapping that assigns each bentity \( \Omega \) in \( \mathcal{A} \) an instance in \( \text{inst}(\Omega) \). Let \( \text{sh}(\mathcal{A}) \) denote all snapshots of \( \mathcal{A} \).

Similar to database updates, three types of bentity updates are considered: insertion, deletion, and modification. Given an enactment \( \sigma \), an insertion, deletion, and modification operation on \( \sigma \) adds a new tuple to a set in \( \sigma \) with its (local) key defined, removes a tuple (in which each set should be empty) from \( \sigma \), and changes the value of a non-key primitive attribute in \( \sigma \). We consider only updates whose result is an enactment. Let \( \delta(\sigma) \) be the result of applying an update \( \delta \) on an enactment \( \sigma \).

Given an enactment \( \sigma \) of a bentity \( \Omega(a, K, L, \text{dep}) \), an update on an attribute \( a \) is eligible on \( \sigma \) if each attribute in the graph induced by \( \text{dep} \) having a path to \( a \) has a non-\( \bot \) value in \( \sigma \). For an enactment \( \sigma \) and an instance \( l \) of a bentity \( \Omega \), the set of all eligible updates on \( \sigma, l \), is denoted as \( \Delta^+_{\Omega, l} \), \( \Delta^-_{\Omega, l} \) (resp.), and conveniently as \( \Delta_{\Omega, l} \) when clear.

As shown in Section II, an artifact is usually associated with a concrete lifecycle. In order to generalize our result, in the following, we abstract different lifecycles into “update patterns” that restrict how artifacts should progress (i.e., update its bentity), such that as long as the lifecycles comply with the specified update patterns, our results in this paper hold.

Given a bentity \( \Omega \), an update class of \( \Omega \), is a set (“insert”, “delete”) \( \times A_s \cup \{\text{“modify”}\} \times A_n \), where \( A_s \) and \( A_n \) are the sets of all set and non-key primitive attributes in \( \Omega \) respectively. Intuitively, an update class is a “template” of possible updates for enactments. Each element in an update class is called an update type.

An “update constraint” is a conjunction of equality or inequality conditions of primitive attributes. If the conjunction is satisfied (by an enactment), all updates of the specified update type are not allowed to apply.

**Definition:** Given a bentity \( \Omega \), an update constraint over \( \Omega \) is of form \( (\beta_{i=1}^n \phi_i) \rightarrow U \), where (1) \( n \in \mathbb{N}^+ \), (2) for each \( i \in [1..n] \), \( \phi_i \) is either \( a_1 = a_2 \) or \( a_1 \neq a_2 \), where \( a_1, a_2 \) are primitive attributes in \( \Omega \) such that there are functional paths from \( a_1 \) to \( a_2 \) as well as \( a_2 \) to \( a_1 \), or values in \( \text{DOM} \), and (3) \( U \) is an update type of \( \Omega \), s.t., there is a functional path from the attribute in \( U \) to each variable in \( \phi_i \) (\( i \in [1..n] \)).

**Example 3.2:** For the bentity in Fig. 2, an update constraint is “\( a\text{Cust}_\text{Last}_\text{Name} = \bot \land a\text{Cust}_\text{First}_\text{Name} = \bot \rightarrow \text{modify}, a\text{Reason} \)” indicating that if a requesting customer has not been determined yet, then reason for the request cannot be filled in. Another example could be “\( 1 \rightarrow \text{update}, a\text{Cust}_\text{Addr} \)”, denoting that \( a\text{Cust}_\text{Addr} \) cannot be updated.

**Definition:** An artifact system \( \Sigma \) is a pair \((\mathcal{A}, \eta)\), where \( \mathcal{A} \) is a bentity system, and \( \eta \) is a mapping from each \( \Omega \in \mathcal{A} \) to a set of update constraints of \( \Omega \). A snapshot of \( \Sigma \) is a snapshot of \( \mathcal{A} \).

Given an artifact system \((\mathcal{A}, \eta)\), an artifact is a pair \((\Omega, C)\), where \( \Omega \in \mathcal{A} \) and \( C = \eta(\Omega) \) is a set of update constraints over \( \Omega \), called update pattern.

Given an enactment \( \sigma \) of \( \Omega \), an update \( \delta \in \Delta^+_{\Omega, l} \) is applicable to \( \sigma \) if \( \delta \) does not violate each update constraint in \( C \) and \( \delta \).
Fig. 4: Process execution and database

is eligible to \( \sigma \). The notion of “applicable” can be naturally extended to the artifact system snapshots. Given a snapshot \( \Pi \) of an artifact system, denote \( \Delta \) to be all the applicable updates to \( \Pi \).

IV. ENTITY-DATA MAPPING RULES

Process data are typically stored in a database (Fig. 4). When a process instance updates its business entity, the corresponding updates on the database should be performed. This section introduces “Entity-Data” mapping rules, or ED rules, a rule-based data mapping language for specifying correspondence between BP enactments and databases, and shows that ED rules are equivalent to a subset of Clio with respect to the expressiveness of mapping databases to enactments.

A business entity can be naturally seen as a “view” on the database, when treating data mapping as queries. In this case, finding database updates for a process update resembles the view update problem studied in the relational databases [3], [9]. There are a few important differences.

First, the view update problem often has no solutions [3], and is hard in restricted cases when solutions exist, the presence of key and foreign key constraints further complicates the problem. However, a process instance acts on one entity instance at a time; even without data modeling as business entities, appropriate database updates are always found during process implementation. It suggests that business entities are more restrictive than views in practice and its update problem is generally solvable.

Second, business entities are hierarchically structured but not relational. Most view mechanisms are not suitable, with an exception of schema language mechanisms such as Clio [11]. In this case, the database is the source and the entity is the target. However, the (view or) target update problem has not been studied. Our path expression based language is restrictive as a query language but more natural for specifying data mappings on process data. More importantly, the language facilitates solutions to the problem of propagating process updates to the database. In § IV-A, we identify a subclass of Clio rules that are equivalent to ED rules.

To define the mapping language, we introduce the notions of “functional multi-paths” (retrieving attributes from another attribute in a business entity), “reference paths” (retrieving attributes from another attribute in a database), “cross-reference paths” (retrieving attributes in a database from an attribute in a business entity), and “key-mapping rules” (matching a (local) key in a business entity with (part of) a key in a database).

For each \( m > 0 \), an \( m \)-ary functional multi-or fun-mpath is an expression of form “\( p_0[p_1, ..., p_m] \)”, where (1) for each \( i \in [1..m] \), \( p_0, p_i \) (concatenation \( p_0 \) and \( p_i \)) is a fun-path and the tail of \( p_i \) is a primitive attribute, (2) the head of \( p_0 \) is primitive, and (3) for each \( i \in [1..m] \), there is a path from the tail of \( p_i \) to the head of \( p_0 \) in graph \( (PM(\alpha), dep) \) (i.e., complying with the access dependencies to prevent that a referenced attribute is undefined). When \( m = 1 \), we may drop “[” “]” for convenience.

Example 4.1: In Fig. 2, an example fun-mpath could be “\( aPart\{aPartID, aReplacement\{Parts, aServiceID\} \} \)”.

Similarly, in the database side, a “reference paths” traverses through a chain of foreign keys references to retrieve a single non-primary attribute. Given a database schema \( (\mathbb{R}, F, A) \) and \( n > 0 \), a reference (or ref-) path is an expression of form \( \exists R_1(k'_1), k_1@R_2(k'_2), k_2@...@R_{n-1}(k'_{n-1}), k_{n-1}@R_n(k'_n) \), where for each \( i \in [1..(n-1)] \), \( k_i \subseteq \text{Att}(R_i) \) and \( (k_i, k_{i+1}) \in F \) (the value of \( k_i \) should be the same as \( k'_{i+1} \)), and \( a \in \text{Att}(R_n) \).

Example 4.2: In Fig. 3, a reference path is “\( \text{tMaterialInfo\{tMaterialID, tServiceID\}_MI, tServiceID\}_MI\{tServiceID\}_SI\{tServiceID\}_SI\{tRepair\{tRepairID\}_RI, tServiceID\}_RI\{tServiceID\}_RI\{tServiceID\}_RI\{tMaterial\}_MI, tMaterial\}_MI \)” that denotes the \( tReason \) value of the tuple in \( tRepair \) corresponding to a unique tuple in \( tMaterialInfo \) by matching foreign keys.

A “cross-reference” path defined below concatenates a fun-mpath and a ref-path to set up a relationship between a non-key attribute in a business entity and an attribute in a database schema. A cref-path defines below concatenates a ref-path and the key of the first relation schema in the ref-path.

Example 4.3: For the database schema in Fig. 3 and the business entity in Fig. 2, a cref-path can be “\( \text{aReason\{aRepair\{Info\, aID \@ tRepair\{tRepairID\}_RI\}, tRepair\{tRepairID\}_RI\} \text{tRepair\{tRepairID\}_RI\} \text{tMaterial\{tMaterialID\}_MI, tMaterial\}_MI \} \)” that denotes the \( tReason \) attribute can be retrieved from the \( aReason \) attribute by matching the values of \( aID \) and \( tRepairID \). A more complicated example is “\( \text{aPart\{aPartID, aReplacement\{Parts, aServiceID\} \}_MI\{tMaterial\}_MI, tMaterial\}_MI \)”.

In our framework, a cref-path establishes the relationship between a non-key attribute in a business entity and an attribute in a database. For attributes in a key or local key, a “key-mapping rule” is used, which establishes the relationship between a key (or local key) in a business entity and a key (resp. part of a key) in a database.

Let \( S = (\mathbb{R}, F, A) \) be a database schema and \( \Omega(\alpha, K, L, dep) \) a business entity. Further suppose that \( \gamma \) is a (local) key of \( \Omega \), \( R \subseteq \mathbb{R} \) a relation schema, and \( k \subseteq \text{Att}(R) \) such that \( |\gamma| = |\gamma| \). A key-mapping rule of \( \gamma \) is an expression of form “\( Rx' \) or “\( R_k WHEN \varphi \)” such that (1) if \( \gamma \) is the ID of \( \Omega \), then \( k \) is a key of \( R \) and the key-mapping rule must have the form “\( \gamma \) | \( Rx' \)”, (2) if \( \gamma \) is not the ID of \( \Omega \), “\( R_k WHEN \varphi \)” must be used; moreover, \( k \) is a key of \( R \)
if $γ$ is a key, (3) if $γ$ is a local key in $Ω$, $κ$ is contained in a key of $R$ and specifically $κ = κ' - \{ f | \forall x \in \text{Keys}(R), (f,k') \in F \}$ where $κ'$ is a key of $R$. (4) $ϕ$ is an expressions with form $p_0[p_1, ..., p_n] = R.κ'$, where $p_0[p_1, ..., p_n]$ is a fun-map, the head of $p_0$ is the first element of $γ$ to denote that the all the referenced attributes (based on $p_0[p_1, ..., p_n]$) should start from $γ$, and $κ'$ is a foreign key of $R$; the tails of $p_1,...,p_n$ (i.e., the referenced attributes from $γ$) form the (local) key in the “upper level”; if $γ$ is the (local) key for a set attribute $a$, and $b$ is an ancestor of $a$ in the entity and a set attribute (or the root) with no set attribute between $a$ and $b$, then the tails of $p_1,...,p_n$ form the (local) key of $b$, and (5) for each $p_0[p_1, ..., p_n] = R.κ'$ in $ϕ$, if $λ(κ')$ is “−” or “+”, then for each $i \in [0..n]$, $p_i$ contains no set attributes; otherwise (i.e., $λ(κ')$ is “*” or “+”), $p_i$ contains a set attribute for some $i \in [0..n]$.

Intuitively, a key-mapping rule defines an equality between a (local) key $γ$ (in a bentity) and (part of) a key $κ$ (in a database). The “WHEN” conditions are needed for the situations when $γ$ equals to $κ$ under a context (presence of a foreign key) either not in a nested set (a key in the bentity) or within a nested set (a local key).

**Example 4.4:** Consider the database schema in Fig. 3 and the bentity in Fig. 2, the key-mapping rule $tRepairID$ for $aID$ denotes that there exists a tuple of $tRepair$ whose value of $tRepairID$ is equivalent to the value of $aID$ in a bentity. Moreover, the key-mapping rule for $aServiceID$ could be “$tServiceInfo.tServiceID WHEN aServiceID.aServiceInfo.aID = tServiceInfo.tRepairID_SI$” to denote that there should exist a tuple of $tServiceInfo$ whose $tServiceID$ value is equal to the value of $aServiceID$ in a bentity only under the circumstance where in the same bentity and the tuple, the value of $aID$ is the same as the value of $tRepairID_SI$.

**Definition:** Given a bentity $Ω(\mathbb{R},\mathbb{L},\mathbb{dep})$, a database schema $S = (\mathbb{R}, F, λ)$, if $γ$ is a (local) key, or a primitive attribute not in a (local) key in $Ω$, an entity-data (ED) mapping rule of $γ$ is either a key-mapping rule if $γ$ is a key or local key, otherwise an expression “$\Leftarrow p^\prime”$ where $p$ is a cref-path whose head is $γ$.

An ED rule for a primitive attribute or (local) key $a$ may have different forms. If $a$ is not a key or a local key, then a cref-path can uniquely match the value of $a$ in a bentity to the value of an attribute in a database (these two values should always be the same). Otherwise, if $a$ is a (local) key, then two cases can be obtained: (1) if $a$ is the ID, then there should exist a key $κ$ in the corresponding relation schema $R$, such that the value of $a$ in a bentity is the same as the value of $κ$ in some instance of $R$. (2) otherwise, the value of $a$ should be “scoped” by the “WHEN” condition.

**Example 4.5:** Fig. 5 shows the mapping rules for some attributes based on Figs. 3 and 2 (not including the attributes in dashed boxes). The meaning of the mapping rules for $aID$ and $aServiceID$ is the same as explained in Example 4.4. And, comparing with Example 4.3, the mapping rule for $aReason$ means that the value of $aReason$ is the same as the value of $tReason$ in a tuple of $tRepair$ whose key ($tRepairID$) has the value of $aID$.

Given a bentity $Ω$ and a database schema $S$, a mapping rule $r$ (with respect to $Ω$ and $S$) for a (set of) attribute(s) $A$ in $Ω$ is satisfied by an enactment $σ \in \text{Ent}(Ω)$ and a database $d \in \text{inst}(S)$, denoted as $(σ, d) \models r$, if one of the following conditions is satisfied:

- $r$ is of form “$R.κ$” and there exists a tuple $τ$ of $R$ in $d$, such that the value of $κ$ in $τ$ is the same as the value of $A$ in $σ$. (Notice that the relationship between the ID $A$ and the key $κ$ is one-to-one.)
- $r$ is of form “$R.κ WHEN \varphi$”, for each tuple $τ$ of $R$ in $d$ that satisfies $\varphi$ wrt $σ$, there exists a value for $A$ identical to the value of $κ$ in $τ$, and for each value $v$ for $A$ in $σ$, there exists a tuple $τ$ of $R$ in $d$ such that the value of $κ$ in $τ$ is $v$ and $τ$ and $σ$ satisfy $\varphi$.
- $r$ is of form “$\Leftarrow p@R_j(κ')\cdots@R_1(κ')$,a^\prime$”, where $p$ is a functional multi-path, for each value of all tails of $p$ in $σ$, there exists a value $v$ for $A$ in $σ$ such that $v$ is the value of $κ'$ in a tuple of $R_j$ that is retrieved according to the cref-path, and vice versa.

**Definition:** Given a bentity $Ω$ and a database schema $S$, a set of ED rules is an ED cover if it contains exactly one mapping rule for each (local) key or non-key primitive attribute of $Ω$.

The set of mapping rules in Fig. 5 is an ED cover for the bentity in Fig. 2 (ignoring the attributes in dashed boxes). The following property confirms that an ED cover yields a well defined data mapping.

**Lemma 4.6:** Let $Ω$ be a bentity, $S$ a database schema, $M$ an ED cover. Then, for each database $d \in \text{inst}(S)$ and a value $v_{Ω(p)}$, there exists at most one enactment $σ \in \text{Ent}(Ω)$ holding $v_{Ω(p)}$ as its ID value such that for each $r \in M, (σ, d) \models r$.

A. Clio and an equivalence result

In this subsection, we view ED rules and Clio [11] as “queries” mapping database instances to enactments, define the notion of “equivalence” of such queries, and then formulate a syntactic subclass of Clio, called “entity maps”. The main result shows that ED covers are equivalent to Clio entity maps.

We fix $S$ to be a database schema, $Ω$ a bentity, and $x_{Ω}$ an enactment ID (used as a variable), unless otherwise specified.

Given an enactment of $Ω$ with ID $x_{Ω}$, an ED cover $M$ specifies all values in the database that correspond to values in the enactment $x_{Ω}$. The rules can also be used to “fetch” the values for the enactment $x_{Ω}$ using the correspondence. For an instance $d \in \text{inst}(S)$, let $M(d)$ be the output enactment $x_{Ω}$.

Clio is a schema mapping language using tgd-like (tuple generating dependency) rules to transform hierarchical source data to hierarchical target data. For our purpose, we consider only Clio rules that map relational databases to hierarchical data representing enactments. In order to compare with ED rules that produce one enactment, we allow the variable $x_{Ω}$ as the only free variable (holding the ID of the resulting enactment). We focus on constant-free Clio rules of the following form with only $x_{Ω}$ occurring free:

$$\forall \bar{x} \Phi_Ω \rightarrow \exists Y \Psi_Ω$$

where $\bar{x}$ is a sequence of first-order variables, $\Phi_Ω$ a conjunction of atomic formulas of form “$R(\bar{x})$” with $R$ a relation in $S$ and $Y$ variables in $\bar{x} \cup \{x_{Ω}\}$, $Y$ variables for (nested) tuple/set constructs in $Ω$ (i.e., $Y$ has no first-order variables), $Ψ_Ω$ a
Fig. 5: Entity-Data mapping rules

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Mapping rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>aID</td>
<td>tRepairInfo.tRepairID</td>
</tr>
<tr>
<td>aReason</td>
<td>aRepairInfo.aID@tRepair(tRepairID).tReason</td>
</tr>
<tr>
<td>aDate</td>
<td>aDate.tRepairInfo.aID@tRepair(tRepairID).tDate</td>
</tr>
<tr>
<td>aCust_Last_Name</td>
<td>aCust_Last_Name.tCust_Name.aCustomer.aID@tRepair(tRepairID).tCustomerLN</td>
</tr>
<tr>
<td>aCust_First_Name</td>
<td>aCust_First_Name.tCust_Name.aCustomer.aID@tRepair(tRepairID).tCustomerFN</td>
</tr>
<tr>
<td>aCust_Addr</td>
<td>aCust_Addr.tCust_Last_Name.aCust_First_Name@tUser(tLastName, tFirstName).tAddress</td>
</tr>
<tr>
<td>aServiceID</td>
<td>tServiceInfo.tServiceID WHEN aServiceInfo.aID = tServiceInfo.RepairID_SI</td>
</tr>
<tr>
<td>aTime</td>
<td>aTime.tServiceID@tServiceInfo(tServiceID).tTime</td>
</tr>
<tr>
<td>(aRP_Last_Name, aRP_First_Name)</td>
<td>tRepairperson.tRepairperson.tRepairID</td>
</tr>
<tr>
<td></td>
<td>aRP_Phone.tRepairInfo.aRP_Last_Name, aRP_First_Name@tUser(tLastName, tFirstName).tPhone</td>
</tr>
<tr>
<td>aPartID</td>
<td>tMaterialInfo.tMaterialID WHEN aPartID.tPart(tMaterialID).tPart == tMaterialInfo.tMaterialID_MI</td>
</tr>
<tr>
<td>aPart</td>
<td>aPart.tMaterialInfo.tMaterialID_MI.tMaterial</td>
</tr>
<tr>
<td>aReviewID</td>
<td>aReviewInfo.aReview_ID.tReview(tReview),tReviewID</td>
</tr>
<tr>
<td>aResult</td>
<td>aResult.tReviewInfo.tReview(tReview).tReviewResult</td>
</tr>
</tbody>
</table>

conjunction of atomic formulas with Ω or variables in Ψ as relations each first-order variable occurs at most once, and $x_0$, occurs in both $Φ_2$ and $Ψ_Ω$. In addition, equality (two occurrences of the same variable) is only allowed between attributes for a foreign key constraint in $S$.

Example 4.7: Equation (**) is a Clio mapping from the relational database in Fig. 3 to the nested structure in Fig. 2. I

Given a set of Clio rules of form (†), we use the semantics similar to the one in [20] that produces a single enactment from a database (rather than the semantics in [10]. In particular, for nested sets, this semantics produces enactments that satisfy partition normal form (PNF) [1].

We introduce further syntactic restrictions on Clio rules. The overall goal is to have a set of Clio rules to map a database to a single bentity instance with the ID $x_0$.

We construct a graph $G_r$ for each rule $r$ of form (†) whose nodes are occurrences of relations in $Φ_r$ and contains edges ($R_1, R_2$) if there is an equality in $Φ_r$ for a foreign key (a) from $R_2$ to $R_1$, or (b) $R_1$ to $R_2$ with a “1-1” constraint.

Condition 1. A rule $r$ is contributing if (1) after collapsing each strongly connected component of $G_r$ into a node, the resulting graph is a tree whose root is the relation containing the variable $x_0$, and (2) each strongly connected component in $G_r$ contains a node whose corresponding formula contains a variable occurring in $Ψ_Ω$ of $r$.

Condition 1 requires that the left-hand side of a Clio rule is connected and join can only happen between keys-foreign keys, thus avoiding cross products.

For each bentity $Ω$, we construct its normalization $Ω^{normal}$ by recursively applying the following operations:

1. Collapse two consecutive tuple constructs,
2. If a set construct $τ$ has a local key within a scope, we duplicate in $τ$ (attributes of) the key of the scope.

Clearly, each attribute in $Ω^{normal}$ corresponds to one attribute in $Ω$ and conversely, each attribute in $Ω$ corresponds to one attribute in $Ω^{normal}$, except for key attributes in a scope of local key(s).

Note that $Ω^{normal}$ has one tuple construct (the root) and 0 or more nested set constructs. For each tuple/set construct $τ$ in $Ω^{normal}$, let $KEY(τ)$ be the set of attributes in the key of $τ$.

Let $Φ_S$ be the set of functional dependencies on $S$ obtained by turning each key, foreign key dependency into a functional dependency and each foreign key with 1-1 constraint into two functional dependencies (both directions). For an atomic attribute $a$ in $Ω$, let $r(a)$ be the set of attributes in $S$ contributing values to $a$, $r(a) = \{ b \mid b$ an attribute in $S$ and there is a variable $x$ corresponding to $a$ in $Ψ_Ω$ and $b$ in $Φ_S \}$. For a sequence $a_1...a_n$ of attributes in $Ω$, let $r(a_1...a_n)$ denote $\cap_{i=1}^n r(a_i)$.

Condition 2. For each tuple/set construct $τ$ in $Ω^{normal}$, the rule $r$ is $τ$-full if (1) each attribute in $KEY(τ)$ corresponds to an attribute (variable) of $Ω$ occurring in $Ψ_Ω$, and (2) there is a sequence $B$ in $r(KEY(τ))$ of attributes of $S$ such that (i) $τ$ has an occurrence $R(τ)$ of a relation $Φ_S$ such that $B$ is a key of $R$ and $τ$ contains variables corresponding to $B$, and (ii) there is a sequence of attributes $C ∈ r(Pm(τ))$ such that $F_τ$ logically implies the functional dependency $B → C$ (recall that $Pm(τ)$ is the set of primitive attributes in $τ$).

A rule $r$ is full if it is $τ$-full for each set/tuple construct $τ$ in $Ω^{normal}$ with an attribute occurring in $Ψ_Ω$.

Condition 2 concerns nested sets: in each nested set, attribute values of each tuple must be uniquely identifiable with the keys in the nested set, avoiding multiple values for a tuple (violating key constraints).

Condition 3. A rule $r$ is consistent if for each pair of constructs $τ_1, τ_2$ in $Ω^{normal}$ where $τ_1$ is a parent of $τ_2$, (1) if $τ_2$ has an attribute occurring in $Ψ_Ω$, so does $τ_1$, and (2) there exist sequences of attributes $B_1 ∈ r(KEY(τ_1)) (i = 1, 2)$ such that (i) $F_τ$ implies the functional dependency $B_1 → B_2$, and (ii) there is an occurrence $R(τ)$ in $Φ_S$ where $τ$ contains both variables for $B_1$ and variables for $B_2$.

Condition 3 insists that each nested set should be connected to its corresponding context (i.e., the parent tuple).

Condition 4. We define $H(r) = \{ τ ∣ τ$ is a tuple/set construct in $Ω^{normal}$ with at least one attribute occurring in $Ψ_Ω \}$. We construct a tree $TQ$ from $Ω^{normal}$ as follows: tuple/set constructs as nodes and child relationships as edges. The rule $r$ is closed if for each $τ ∈ H(r)$, every ancestor of $τ$ in $TQ$ is also in $H(r)$.

Condition 4 concerns the right-hand side of a Clio rule: if a rule produces a value for some nested set in a bentity, all of its ancestors should be present (to provide the context).

Definition: A set $π$ of Clio rules of form (†) is an entity map if (1) every rule in $π$ is contributing, full, consistent, and closed, (2) for each path $p$ in $TQ$ from the root, there is a rule $r ∈ π$ such that $H(r) = \{ τ ∣ τ$ is on $p \}$, and (3) for each pair of
rules \( r_1, r_2 \in \pi, H(r_1) \subseteq H(r_2) \) implies the existence of a 1-1 embedding of \( r_1 \) into \( r_2 \).

In general, Clio rules may produce enactments with undefined (or null) values. To simplify our presentation, we focus only on databases and enactments with no undefined values.

**Definition:** Let \( S \) be a database schema, \( \Omega \) a bentity, \( \pi \) a Clio entity map, and \( M \) a DE cover. Then, \( \pi \) and \( M \) are equivalent, denoted as \( \pi \equiv M \), if for each database \( d \in inst(S) \), \( \pi(d) = M(d) \) when \( d, \pi(d), M(d) \) contain no undefined values.

**Lemma 4.8:** Let \( S \) be a database schema without “0-1” constrained foreign keys and \( \Omega \) a bentity. If \( \pi \) is a Clio entity map and \( M \) a ED cover, then for each \( d \in inst(S) \) without undefined values, neither \( \pi(d) \) nor \( M(d) \) have undefined values.

**Theorem 4.9:** Let \( S \) be a database schema without “0-1” constrained foreign keys and \( \Omega \) a bentity. For each Clio entity map \( \pi \), there is an ED cover \( M \) such that \( \pi \equiv M \). And the converse also holds.

### V. Updatability

This section studies the “updatability” of the ED mapping rules. An “updatable” mapping is able to capture the updates on each bentity enactment and propagate the changes to the database, and vice versa. Consider the framework in Fig. 4, if an update is applied on process instance \( 1 \), then the changes should be reflected in the database. Similarly, if the database is updated, then the corresponding modification should be propagated to each affected process instance. As illustrated in Section II, not every mapping is “updatable”.

Although the ED rules are used to specify corresponding data entries in a database for primitive attributes in a bentity, it is more convenient to define and study mappings from databases to enactments since a database may contain more data entries while every attribute (primitive or complex) in an enactment is stored in the database.

Recall that \( \text{DOM} \) is the domain for all primitive attributes. A permutation of \( \text{DOM} \) is a 1-1 and onto mapping from \( \text{DOM} \) to \( \text{DOM} \). Permutations are naturally extended to tuples, relations, databases, and complex attributes.

**Definition:** Let \( S \) be a database schema and \( \Omega \) a bentity. A partial mapping \( \mu \) from \( \text{inst}(S) \times \text{DOM} \) to \( \text{Ent}(\Omega) \) is a database-enactment mapping (or DE-mapping) if for each permutation \( \pi \) of \( \text{DOM} \), each \( d \in \text{inst}(S) \), and each \( v \in \text{DOM} \), the following conditions hold whenever \( \mu(d, v) \) is defined: (a) \( \mu(\pi(d), \pi(v)) = \pi(\mu(d, v)) \), and (b) \( v \) is the value for the ID attribute in the enactment \( \mu(d, v) \).

Note that in the above definition, a DE-mapping takes a database and a value as inputs and produces an enactment whose ID is the input value. Condition (a) is also referred as “genericity” in the study of database queries [6], which essentially forbids manipulations on values (e.g., concatenations, arithmetic, etc.) and forces the mapping to treat values as (uninterpreted) symbols.

Let \( S \) be a database schema, \( \Omega \) a bentity, and \( M \) an ED cover. Define the mapping \( \mu_{tr} \) from \( \text{inst}(S) \times \text{DOM} \) to \( \text{Ent}(\Omega) \) as \( \mu_{tr}(d,v) = \sigma \in \text{Ent}(\Omega) \) if \( \sigma \) has its ID value \( v \) and \( (\sigma,d) \vdash r \) for each \( r \in M; \mu_{tr}(d,v) \) is undefined otherwise. The following is a consequence of Lemma 4.6.

**Lemma 5.1:** For each database schema \( S \), each bentity \( \Omega \), and each ED cover \( M \) of mapping rules, \( \mu_{tr} \) is a DE-mapping.

We now define the central notion of “updatability” of this section. Roughly speaking, a DE-mapping is database-updatable if every update \( \delta' \) on a database \( d \) corresponds to an update \( \delta' \) on an enactment \( \sigma \) such that the DE-mapping is preserved, i.e., updating the database \( d \) with \( \delta' \) and then applying the DE-mapping is identical to applying the DE-mapping first followed by the update \( \delta' \) on \( \sigma \). The converse direction is bentity-updatability. A slight technical problem is that each update \( \delta' \) often corresponds to a sequence of updates on the enactment (resp. database). Thus the following definition allows sequences of updates.

**Definition:** Let \( S \) be a database schema, \( \Omega \) a bentity, \( \Delta_s \subseteq \Delta_S \) and \( \Delta_o \subseteq \Delta_O \) classes of updates on \( S \) and \( \Omega \) (resp.). A DE-mapping \( \mu \) is said to be
- **database-updatable with respect to** \( \Delta_s, \Delta_o \) if for each database update \( \delta' \in \Delta_s \), there is a sequence \( \delta' \) of bentity updates in \( \Delta_o \) such that for all \( d \in \text{inst}(S) \) and \( v \in \text{DOM} \), \( \mu(\delta', d, v) = \delta'(\mu(d, v)); \)
- **bentity-updatable w.r.t.** \( \Delta_s, \Delta_o \) if for each bentity update \( \delta' \in \Delta_o \), there is a sequence \( \delta' \) of database updates in \( \Delta_s \) such that for all \( d \in \text{inst}(S) \) and \( v \in \text{DOM} \), \( \mu(\delta', d, v) = \delta'(\mu(d, v)); \)
- **updatable w.r.t.** \( \Delta_s, \Delta_o \) if it is both database-updatable and bentity-updatable w.r.t. \( \Delta_s, \Delta_o \).

Updatability states that it is the same effect whether to apply an update followed by a mapping or a mapping followed by an update. With updatability, each update on databases can be propagated to bentities, and vice versa.

The notion of DE-mapping is similar to the schema mapping in [11]; however, the schema mapping studies did not concern updates. For bentity-updatability, it is similar to view updates [3], [9] in relational databases; however, view updates focus on relational models and bentity-updatability focuses on hierarchical models. Further, it is not clear if the view complement approaches [3], [17] can be adopted in this work. Updates on XML views over relational databases were studied in [26] with a focus on complexity and/or testing whether an XML view update can be translated. In comparison, DE-mappings correspond to very restricted views and we focus on sufficient conditions to ensure that updates can be translated.

**Theorem 5.2:** For each database schema \( S \), each bentity \( \Omega \), and each ED cover \( M \), the DE-mapping \( \mu_{tr} \) is database-updatable with respect to \( \Delta_s \) and \( \Delta_o \).

Theorem 5.2 is easy to prove. Since an ED cover identifies a database attribute (value) for each attribute (value) in a bentity, the mapping can be easily expressed as a database query. Therefore database updatability follows immediately. However, Theorem 5.2 fails for bentity-updatability, which was illustrated in Example 2.1.

Given \( r \) as a mapping rule for some primitive attribute or (local) key \( y \) in a bentity, if \( r \) is of form “\( R.k\ (\text{WHEN} \ \varphi) \)”
then each primitive attribute in $\gamma$ is said to be associated with each attribute in $\kappa$; otherwise, if $r$ is of form \( \bowtie p_0, [p_1, \ldots, p_n] \) \( \bowtie R_j(k_1), k_1 \bowtie \ldots \bowtie R_m(k_n), a \), then $\gamma$ is associated with $a$.

Let $M$ be an ED cover for a database schema $S$ and a bentity $\Omega$. Two primitive attributes in $\Omega$ are overlapping if the two sets of database attributes that they are associated with (resp.) are not disjoint.

**Theorem 5.3:** For each database schema $S$, each bentity $\Omega$, and each ED cover $M$, if primitive attributes in $\Omega$ are pairwise non-overlapping, then the DE-mapping $\mu_\Omega$ is bentity-updatable with respect to $\Delta_\Sigma$ and $\Delta_\Omega$.

**Corollary 5.4:** Let $S$ be a database schema, $\Omega$ a bentity, $M$ an ED cover, and $\Delta_\Sigma$ and $\Delta_\Omega$ subsets of updates on $S$ and $\Omega$ (resp.). The DE-mapping $\mu_\Omega$ is updatable with respect to $\Delta_\Sigma$ and $\Delta_\Omega$ if either (1) primitive attributes in $\Omega$ are pairwise non-overlapping, or (2) $\Delta_\Omega$ contains no updates on any overlapping attribute.

The set of mapping rules in Fig. 5 is updatable as all the primitive attributes in $\mu$-mapping (Fig. 2) are pairwise non-overlapping.

Finally, we remark that since ED covers correspond to Clio entity maps, the updatability results on ED covers can be extended to entity maps informally. But formal statements are problematic due to presence of undefined values. Nevertheless, entity maps can always be converted into ED covers for implementation of BP data accesses.

**VI. ISOLATION**

In this section, we study the notion of “isolation” that prohibits any situations when two bentities in a system update their attributes that are mapped to the same entry in a database. Consider the framework in Fig. 4, if process instances 1 and 2 apply updates on attributes that are mapped to the same entry in the database, then these two instances are “affecting” each other, which may not be intended for the process design.

Section V focuses on DE-mappings that result in individual enactments. In this section, we study mappings from databases to shapshots. We fix $S$ to be an arbitrary database schema and $A$ an arbitrary (bentity) system, unless otherwise indicated.

**Definition:** A mapping $\mu$ from inst$(S)$ to sh$(A)$ is a database-snapshot (or DS-mapping) if for each permutation $\pi$ of Dom, each $d \in$ inst$(S)$, $\mu(\pi(d)) = \pi(\mu(d))$.

DE-mappings and DS-mappings are closely related. For each DS-mapping $\mu$ and each bentity $\Omega \in A$, we define a mapping $\mu_\Omega$ derived from $\mu$ as follows. For each database $d \in$ inst$(S)$ and each $v \in$ Dom, $\mu_\Omega(d, v) = e$ if $e$ is an enactment in $\mu(d)$ with its ID value $v$, undefined otherwise. Let $DE(\mu) = \{ \mu_\Omega | \Omega \in A \}$ be the set of mappings derived from $\mu$. Conversely, for each set $\mu_\Omega = \{ \mu_\Omega | \Omega \in A \}$ of DE-mapping, we define the mapping DS$(\mu_\Omega)$ as: $DS(\mu_\Omega)(d) = \bigcup_{\Omega \in A} \mu_\Omega(d, v) | v \in$ Dom and $\mu_\Omega(d, v)$ is defined.

**Lemma 6.1:** Let $\mu_\Omega = \{ \mu_\Omega | \Omega \in A \}$ be a set of DE-mappings and $\mu$ a DS-mapping. Then $DE(\mu)$ is a set of DE-mappings and $DS(\mu_\Omega)$ is a DS-mapping.

**Definition:** A DS-mapping $\mu$ is bentity-updatable if each DE-mapping in $DE(\mu)$ is bentity-updatable.

In the remainder of this section, we only focus on bentity-updatable DS- or DE-mappings. While the Lemma 6.1 states that DE-mappings can be naturally extended to DS-mappings, and vice versa, additional issues may arise due to “conflicting” updates by two enactments. In some cases, two updates upon two separate enactments (possibly of the same bentity) might be propagated to the same attribute in the same tuple in a database. Under such circumstances, maintenance of bentities as well as the databases becomes more complicated, and their semantic implications may not be clear. In our formal analysis, we intend to identify DS-mappings that do not have such conflicts, which may serve as a guideline for design or the execution engine to manage executions.

Let $a$ be a complex attribute, $K$ a set of keys and local keys in $a$, and $A$ a (possibly infinite) set of primitive attributes. Let $\kappa$ be the set of all primitive attributes in (local) keys in $K$. The projection of $a$ on $\kappa$, denoted as $\Pi_a(\kappa)$, is a complex attribute with all primitive attributes not in $A \cup \kappa$ removed. The projection operation is naturally extended to values of attribute $a$. We extend projection to $\Omega$ and enactments in $Ent(\Omega)$, $A$ and snapshots in sh$(A)$ naturally. Given updates $\Delta \subseteq \Delta_\Omega$, let $W(\Delta)$ be the set of all attributes updated by operations in $\Delta$.

**Definition:** Given $S$ as a database schema, $\Sigma = (A, \eta)$ an artifact system, and $\mu$ a DS-mapping from inst$(S)$ to sh$(A)$, $\mu$ is isolating w.r.t. $\eta$, if for each $d \in$ inst$(S)$ and each bentity update $\delta' \in \Delta^{ed}$, there is a sequence $\delta^d$ of database updates in $\Delta_\Sigma$ such that $\Pi_{W(\Delta^{ed})}(\mu(\delta^d(d))) = \Pi_{W(\Delta^{ed})}(\delta(\mu(d)))$.

An isolating DS-mapping prohibits the situation where two running bentity enactments can both update two attributes respectively that are mapped to the same entry in the same tuple in a database, hence, “affecting” each other in a snapshot. Isolation is needed for the ScGA tool [23] to work properly.

**Example 6.2:** In Fig. 2, if an aRepair enactment updates the customer address (aCust_Addr), it may affect some other enactment(s), e.g., of Customer Info Review, that may also be able to update the customer address. Therefore, the underlying DS-mapping is not isolating.

Notice that it is not undesirable if a DS-mapping is not isolating. Isolation is only to provide a guideline for the process designers to know that two bentities may interfere each other. By understanding the interference, a designer can add update constraint to prevent such situation when necessary.

**Example 6.3:** Continue with Example 6.2. A designer can add update constraint “$1 = 1 \rightarrow$ (update, aCust_Addr)” to prevent an aRepair enactment from updating the customer addresses, which is intuitive, as the repair process only serves for the repair purpose.

**Theorem 6.4:** Let $S$ be a database schema, $(A, \eta)$ an artifact system, and $M$ be a set of ED mapping rules whose derived DS-mapping $\mu_\Sigma$ (from inst$(S)$ to sh$(A)$) is bentity-updatable. Then it can be determined in exponential time, whether $\mu_\Sigma$ is isolating with respect to $\eta$.

To prove Theorem 6.4, an algorithm is provided. The main idea is that for each pair of bentities in a system, first compute
a “conflict set”, whose elements are pairs of attributes that are mapped to the same attribute in database. Then is to run a symbolic execution to check if two updates of two bentity enactments can be applied to the corresponding conflict sets.

In general, a non-empty conflict set for two bentities does not imply that corresponding mapping is not isolating. If no two updates can be applied during the execution upon each pair of elements (resp.) in each conflict set, the mapping is still isolating.

Before computing the “conflict sets” (for two bentities), we first introduce the notion of “nonconflicting” for mapping rules, such that if a database attribute is mapped by such a mapping rule, then this attribute can only be “accessed” by its corresponding primitive bentity attribute or (local) key.

Given a bentity $\Omega$, a database schema $S$, and an updatable mapping rule set $M$ (with respect to $\Omega$ and $S$), a mapping rule $r \in M$ (for a (local) key or a primitive attribute $a$ in $\Omega$) is nonconflicting if one of the following conditions holds.

- If $r$ is a key,
- If $r$ is of form “$R, a$ WHEN $\varphi$”, then there exist some tails of some functional multi-paths $\varphi$ that form a key, or
- If $r$ is of form “= $p_0[R(k'_1), k_1]@R_1(k'_2), k_2]@...@R_m(k'_m), a$”, then some tails of $p_1, ..., p_n$ form a key, and for each $i \in [1..(m-1)]$, either $k'_i \subseteq k_i$, or $\lambda(k_i)$ is “??” or “+”.

**Example 6.5:** Based on the mapping rules shown in Fig. 5, the mapping rule for attributes “aCust_Addr” and “aRP_Phone” are not nonconflicting; and the mapping rules for other attributes are nonconflicting.

**Definition:** Given two bentities $\Omega_1$ and $\Omega_2$ a database schema $S$, and two updatable mapping rule sets $M_1, M_2$ (with respect to $\Omega_1$ and $S$, $\Omega_2$ and $S$ respectively), the conflict set of $\Omega_1$ and $\Omega_2$, denoted as $cf(\Omega_1, \Omega_2)$, is a set of pairs of non-key primitive attributes or (local) keys, such that for each $(a_1, a_2) \in cf(\Omega_1, \Omega_2)$, (1) $a_1$ and $a_2$ have mapping rules in $M_1$ and $M_2$ respectively, (2) both the mapping rules for $a_1$ and $a_2$ are not nonconflicting, and (3) the two (sets of) attributes associated with $a_1$ and $a_2$ have overlapping.

**Example 6.6:** Suppose a system only contains two bentities, namely aRepair1 and aRepair2, whose structures are exactly the same and shown in Fig. 2. Suppose both of them adopt the mapping rules defined in Fig. 5. The conflict set of aRepair1 and aRepair2 only contains two pair (aCust_Addr, aCust_Addr) and (aRP_Phone, aRP_Phone).

Intuitively, a conflict set denotes all the possible attributes in two bentities that can be mapped into the same attribute in the same tuple in a database during the execution.

Given a bentity $\Omega(a, K, L, dep)$, a symbolic bentity of $\Omega$ is a complex attribute $\tilde{\Omega}$, where $\tilde{\Omega}$ is obtained from $\Omega$ by replacing each set attribute $a$: $(a_1, ..., a_n)$ by a tuple attribute $\tilde{a}$: $(a_1, ..., a_n)$.

Given a bentity $\Omega$ and a set of update constraints $C$ of $\Omega$, for each $c \in C$, denote $N(c)$ to be the number of distinct (primitive) attributes and values in $c$. Let $N(C)$ be the largest $N(c)$ for each $c \in C$. Then the (symbolic) domain of $\Omega$ is $\{v_1, v_2, ..., v_{N(C)}, \perp\}$, where for each $i \in [1..N(C)]$, $v_i$ is a distinct “defined” value.

A symbolic enactment $\tilde{\sigma}$ of a symbolic bentity is an assignment of each primitive attribute in $\tilde{\sigma}$ to an element in the symbolic domain; and $\tilde{\sigma}$ should comply with the access dependency, i.e., an attribute can have a defined value only if each attribute it depends on has a defined value. Notice that given a symbolic enactment $\tilde{\sigma}$ with domain $\{v_1, v_2, ..., v_{N(C)}, \perp\}$, it is sufficient to check if $\tilde{\sigma}$ can satisfy each constraints in $C$.

For reading convenience, for each attribute, set of attributes, key, or local key $k$ in a bentity $\Omega$, we denote the corresponding “attribute”, “set of attributes”, “key”, or “local key” of $k$ as $\tilde{k}$ in its corresponding symbolic bentity $\tilde{\Omega}$.

Similar to the “insertion”, “deletion”, and “modification” updates for a bentity, a symbolic bentity can also have these three updates as well. More specifically, given a bentity $\Omega$, a symbolic bentity $\tilde{\Omega}$ of $\Omega$, and a set of update constraints $C$ of $\tilde{\Omega}$, an insertion of a complex attribute $\tilde{a}$ (based on a symbolic enactment of $\tilde{\Omega}$) is to assign a set of child attributes $\tilde{K}$ of $\tilde{a}$ from $\tilde{\Omega}$’s to elements in $\{v_1, ..., v_n\}$, such that $\tilde{a}$ is a set attribute and $\tilde{K}$ forms a (local) key for $\tilde{a}$ in $\tilde{\Omega}$; a deletion is to assign each attribute in a tuple to $\perp$; and a modification is to assign a non-key primitive attribute to an element in $\{v_1, ..., v_n\}$.

An update is applicable with respect to a symbolic enactment and a set of update constraints, if no constraint prohibits the applying of the update and after the update, the result assignment still forms a symbolic enactment.

Based on a symbolic enactment, a finite set of “symbolic updates” can be computed. Given bentity $\tilde{\Omega}$, a symbolic bentity $\Omega$ of $\tilde{\Omega}$, an update pattern $C$ of $\tilde{\Omega}$, and a symbolic enactment $\tilde{\sigma}$ of $\tilde{\Omega}$, a symbolic update set $\Delta(\tilde{\sigma}, C)$ is a set of all applicable insertions, deletions, and modifications based on $\tilde{\sigma}$.

Given two bentities $\Omega_1$ and $\Omega_2$ a database schema $S$, and two updatable mapping rule sets $M_1, M_2$ (with respect to $\Omega_1$ and $S$, $\Omega_2$ and $S$ respectively), $\Omega_1$ and $\Omega_2$ are independent (with respect to $M_1$ and $M_2$) if and only if for each pair of symbolic enactments $\tilde{\sigma}_1$ and $\tilde{\sigma}_2$ of $\Omega_1$ and $\Omega_2$ (resp.), each $\delta_1 \in \Delta(\tilde{\sigma}_1, C)$, $\delta_2 \in \Delta(\tilde{\sigma}_2, C)$, and each pair of attributes $(\tilde{a}_1, \tilde{a}_2)$ updated by $\delta_1$ and $\delta_2$ (resp.), there does not exists a pair $(\tilde{a}_1, \tilde{a}_2) \in cf(\tilde{\Omega}_1, \tilde{\Omega}_2)$, such that $\tilde{a}_1$ occurs in $\tilde{\sigma}_1$ and $\tilde{a}_2$ occurs in $\tilde{\sigma}_2$.

A DS-mapping is isolating if and only if each pair of (possibly the same) bentities in the system is independent (with respect to their corresponding mapping rules).

**Example 6.7:** Continue with Example 6.6; the mapping from the database in Fig. 3 to this system is isolating with respect to constraints “$1 = 1 \rightarrow (\text{update, aCust_Addr})$” and “$1 = 1 \rightarrow (\text{update, aRP_Phone})$”.

The complexity to determine the conflict sets is polynomial with respect to the size of the database schema, bentities, and mapping rules. While the complexity of the symbolic execution is exponential with respect to to the size of all the bentities.

Notice that the algorithm described above is to check the “write-write” conflicts within a system. It is straightforward to adapt this algorithm to check the “write-read” conflicts, i.e., an update upon an attribute within an enactment will not affect the value of each other enactment in the system.

Similar to the remark at the end of Section V, the isolation results on ED covers can be extended to entity maps.
VII. RELATED WORK

The technical results in the paper are closely related to schema mappings and view updates, and also data integration.

Schema mapping techniques are used in schema/data integration and data exchange between different schemas. The focus there is to reason about and query generated target instance(s) through mapping rules and source instance(s). A classic representative is Cléo [11], [10]. In our entity-database mappings, a relational schema is associated with a hierarchical business entity. The purpose of the mappings is to facilitate data access in business processes, e.g., automated code generation. Updatability turns out to be crucial for maintain data consistency and connectivity between business processes and the enterprise database. Updatability was not studied in conjunction with schema mappings, and updatability is not always possible for schema mapping rules. Since our mapping rules corresponds to a special subset of Cléo mappings, updatability results generalize to these Cléo mappings.

The comparison of bentity updatability and view updates on relation database/XML ([3], [9], [17], [26]) has been addressed in Section V. For database updatability, it is related to view self-maintenance for data warehouses with materialized views. Although not every data warehouse is self-maintainable in general [13], ED covers are always database updatable.

Schema/database integration is achieved through a global schema and mappings between the global schema and local schemas [18]. Earlier work used the relational setting and focuses on query answering [18]. Our ED rules differ from both GAV and LAV and focus on updates.

Our isolation notion is new. The isolation property in database transactions [12] is only marginally related. There is a lack of general understanding for process transactions. Our isolation notion may provide a useful starting point.

BP modeling has been studied intensively for the last two decades [24]. Most technical results are based on process/activity centric models. In recognizing importance of data involvement in BP modeling, artifact/object centric BP models are developed [14]. However, modeling of the linkage between BP data with persistent data in database was never studied.

VIII. CONCLUSIONS

This paper initiates a study on data mappings between BPs and databases through formalizing the data models and formulating a mapping language. This idea of bridging BPs and databases has a potential to allow management issues to be dealt with separately for BPs and for databases while making sound design decisions. For BPM, it allows many interesting problems to be studied in the presence of data, e.g., process evolution. For databases, it brings a new dimension, i.e., BPs, into the database design, in particular, by including BPs’ data needs, database design could avoid problems such as missing data or mismatched semantics.

On the technical front, there are many interesting problems to be addressed. A better understanding is needed for specifying entity-data mappings, alternative languages and relaxing the attribute-attribute mapping requirement are worth considering. Concerning data mapping properties, are there general requirements other than updatability and isolation? For example, general integrity constraints in databases might add more difficulties to updatability. The problems are more interesting to be studied along with lifecycle for bentities, e.g., the issue of BP independence could cleanly separate the “footprints” on data by two BP executions.

REFERENCES