

Structurally Recursive Patterns in Data Modeling and their Resolution

András J. Molnár^[0000–0002–2194–0320]

Christian-Albrechts-University Kiel, Computer Science Institute,
D-24098 Kiel, Germany
`ajm@informatik.uni-kiel.de`

Abstract. By allowing relationship types defined on top of relationship types, the Higher-Order Entity Relationship Model (HERM) enables modeling of complex conceptual structures in a layered way, which usually results in a more compact design than the traditional Entity-Relationship model. Identification of data instances is achieved by composition of the (inherited) key attributes of the referenced instances (foreign keys becoming part of natural keys of higher-order relationships). Well-formedness excludes cycles in the structure. In this paper, we look at the possibility to relax this by allowing structural recursion in the conceptual model. Although it is formally represented as a cycle in the type structure, it will not allow any cycle on the instance (data) level. After looking at some motivating cases and conventional alternatives to this design, and conditions when such a modeling decision is reasonable, we will show how a structurally recursive HERM model can be transformed into an equivalent, conventionally well-formed HERM model, with the utilization of list type construction and complex, variable-length key domains. The analysis reveals some non-trivial aspects of the way of transformation, which may be easily overlooked, but are important to formulate a translation rule for the general case, where derived attributes and aggregation needs are present. The result can be used as a rule for conceptual schema translation for structuring, to obtain an intermediate schema ready for further optimization, and eventually, code generation.

Keywords: Conceptual Modeling · HERM · Structural Recursion · Schema Translation.

1 Introduction and Motivation

The process of designing the database structure of an information system involves modeling and schema design in different levels and stages. Ideally we start with a conceptual model which represents the application domain in an adequate and dependable way [13]. This is translated and optimized in further steps until a well-established logical and physical schema is reached. This process is supported by rules and practices such as schema translation or normalization procedures [5, 6, 2, 3, 7]. While some of these can be applied without any user

input, modeling usually has to involve manual design considerations in order to achieve an appropriate and effective schema, especially in complex systems.

The modeling-to-programming initiative aims to elaborate on methods and tools which can ease this process for schema designers by automatic schema translation, optimization and code generation, and eventually, making the models themselves executable [16]. One promising way to achieve this is to develop and define high-level concepts for enabling explicit user input on how a model should be compiled or interpreted (and run), which then can be specified as directives similarly to a program-code compiler. The model compiler or execution engine can take into account these user-specified guidelines or directives to guide schema translation where multiple options exist.

In this paper, we look at the case of designing a conceptual schema with structural recursion. It occurs wherever a type is constructed so that its data instances may be built on or composed of other data instances of the same type (we assume a database instance in general containing all actual data elements for each type, and call the data elements of a type *data instances* of that type). It is formally represented as a cycle in the type structure, therefore, it is not considered to be a well-formed schema design and not supported by schema translation methods. However, it does not mean any cycle on the instance (data) level. Our aim is to allow such structures by extending the modeling language and translate them into traditionally well-formed, recursion-free schemata, so that it can be further optimized and managed in the standard way. While elaborating this special case has already its contributive value in itself, it can be treated as a case study allowing us to make in-depth considerations as well on what to look for when considering a general schema translation or model execution methodology.

The structure of the paper is as follows: Sect. 2 gives the foundations of the used higher-order entity-relationship (HERM) modeling language with some motivating cases and concepts related to schema-translation and equivalence, considers structural object-oriented design patterns and their translatability to HERM, aspects and examples of structural recursion. The general pattern of structurally recursive schemata to be resolved by translation is presented in Sect. 3 with a generic definition of aggregation, whose proper translatability ensures the completeness of information and equivalence of the resulting schema for data instances composed by structural recursion. Sect. 4 gives the possible schema translations for various settings, briefly showing their development and equivalence with the original schema, along with general guidelines that may guide the translation process as directives. Application of these different translation patterns is summarized in Sect. 5 as an algorithm-like rule. Sect. 6 concludes with some open issues.

2 Foundations and Related Work

2.1 Conventional and Higher-Order Entity-Relationship Models and Our Notations

In conventional *Entity-Relationship (ER)* modeling, a conceptual schema is composed of *entity* and *relationship* types with their *attributes* [4]. Entities (data instances) are subsistent, i.e. they exist on their without depending on other data instances. Relationships connect two or more entity classes and their data instances are dependant on the entities they connect. The connected entities can have named roles in the relationships, and even the same entity type can be connected by the same relationship multiple times, with different roles. In the relational paradigm, set semantics is assumed, however, tables function as multisets in real-life database systems.

Based on the ER model, the Higher-Order Entity Relationship Model (HERM) [14, 15] allows the schema designer to define higher-order relationship types, i.e. relationship types on top of other relationship types and entity types. This way, modeling of complex conceptual structures is possible in a layered way, which in many cases results a more compact schema design than the traditional Entity-Relationship model.

Entities are uniquely identifiable by one or more designated *primary key* attributes. This key is also called a *natural key*, since it reflects how entities are identified in their real-life application domain. For technical reasons, a *surrogate key* without any specific semantics can be introduced in the schema design process, although it is not considered as best practice in the conceptual modeling phase [15]. Relationships usually inherit primary key attributes from the entities they connect. The primary key of a relationship is - by default - composed of these inherited key attributes of the participating entities, optionally extended by own primary key attributes. In the latter case, the same entities can be related to each other multiple times even with set semantics, with different values on the extending key attributes of the relationship. The ER or HERM schema can be translated and optimized to a logical database schema [15].

Identification of data instances of higher-order relationships is by design well-defined as the composition of the (inherited) key attributes of the referenced entity or relationship data instances. This way, a key attribute of an entity type is being inherited by the first-order relationship types directly referring to this entity type, and subsequently, every further higher-order relationship types built on top of them, unless the composed key is explicitly overridden by a relationship type definition.

We are going to use the following, simplified conventional notations when presenting and discussing schemata: entity or relationship types are denoted by capital letters, while attributes and relationship roles are referred by small caps, data instances (single tuples in database instances) by small greek letters. Assuming a particular database instance of a schema, type names denote actual relations as well and their elements are the single data instances (tuples) of the type in the actual relation, denoted by the \in operator. For example, if R is a

relationship type, then $\rho \in R$ is a data instance (tuple with attributes) of it. The dot operator is used to refer an attribute, or, in a case of a relationship, a related member of the connected type with the specified relationship role. If R is defined to relate types P and Q with roles p and q , respectively, then $R.p$ means a function to be applied over any R -instance $\rho \in R$, resulting the P -instance $\pi \in P$ identified by the foreign key referenced by ρ through its relationship role p . The latter function can be inverted using a superscript $^{-1}$, denoting the relationship instance which has the given data instance in its specified role. In this example, the function $P.(R.p)^{-1}$ over P -instances results ρ for π . Furthermore, \bowtie denote a (natural) join of two relations (or types - as a function over their data instances) and P_i denotes the projection of a relation to its specified attributes, e.g. $\Pi_p R$ is the (foreign) key of $R.p$, while $\Pi_{Attr(P)} R \bowtie P$ is the same as $R.p$ if $Attr(P)$ denotes the attributes of P .

Although multiple graphical ER representations exist, we will use the one which was taken and generalized by HERM: rectangles denote entities, diamonds denote relationships. If we use both a diamond and a rectangle for a type, it means it can be either one of them. In HERM, arrows do not represent cardinality or control, but dependence based on structural composition: a relationship depending on entities has arrows towards those entities, while a higher-order relationship depending on other relationships has arrows towards those lower-order relationships it depends on. Cardinality is expressed by number intervals with participation semantics [15], i.e. a number at an entity written towards a relationship shows how many times each data instance of the entity type should (and can) participate in that relationship (minimally and maximally). The default cardinality is $(0, n)$, meaning there is no restriction in the participation.

A special construction of HERM is the *cluster type*, which is a disjoint union of two different (sub)types, and acts as a generalization of these (sub)types [15]. The cluster type is a higher-order relationship, meaning that it is not subsistent on its own, but is dependant on the (sub)types it connects.¹ The cluster type is depicted with a cross in a circle on schema diagrams. We assume a default cardinality of the cluster type as $(0, 1)$, so a specific subtype instance takes part of the cluster type at most once.

HERM defines *type constructors*, by which complex attribute domains or relationship roles can be constructed [15]. For our case, the *list constructor* is relevant, meaning that an attribute of a single data instance of a type can take multiple values in an ordered (indexed) way. It can be applied to relationships as well, resulting a relationship having a list of related data instances of one type or role instead of one single data instance. It is denoted by square brackets on schema diagrams. Key inheritance extends to type constructors in a natural way, when a list-constructed attribute is part of a key, or key attributes are inherited

¹ Specialization is considered in HERM as a different case where the general type is subsistent, and its subtypes depend on it. To model this, the so-called *unary relationships* are used, which is not directly relevant for this paper.

to relationships via list-constructed roles, the list itself is becoming (part of) the (natural) key, resulting a key with a complex domain.

The classical definition of well-formedness of such a schema includes a strict layering requirement, where no cycles are allowed in the structure of these type definitions. We will look at the possibility to relax this by allowing a special case of structural recursion in the conceptual model, using the cluster type construct. Specific cases of its schema translation will use the list type constructor, including inheritance of key attributes, thus resulting a complex key domain for the data instances composed by structural recursion.

2.2 Modeling and Design. Schema Translation and Optimization

Modeling and schema design is a complex process when done systematically, involving different aspects and levels. A common method is to start with a conceptual model of the application domain, which is then refined and adapted to the system needs, translated to a logical schema (relational or other) and implemented in a specific way (physical schema). Optimizations and variations can be done in any of the stages [7]. Modeling-to-programming tries to automate this process as much as possible, while modeling-as-programming aims to make the models themselves executable. Both cases require intermediate schemata and inner optimizations, which can have various influential factors. For instance, although the theory of normalization [5] may be used for automatic schema optimization, there are specific aspects which have a counter-effect and the end (denormalization, [6]), and the schema designer must make considerations towards the optimal schema in a specific case. Some of these can already take effect in the conceptual level, not in the logical schema for which normalization is usually defined [7].

In each of the schema translation steps, it is important to ensure an equivalence of the resulting schema with the original. In most cases it is a simple correspondance between types and attributes, in more complex cases types can be split or merged, rearranged, etc. A bi-directional mapping has to be given which ensures information equivalence, so both schemata have the same expressivity. We will consider information equivalence between the original and translated schemata similarly to the concept of infomorphisms [9, 10], but in most cases the mappings will be obvious.

2.3 Structural Design Patterns and their Translatability

Design patterns are investigated in conceptual and relational modeling [7, 1], but they are mostly well-known in object-oriented modeling and design [8]. Structural patterns include the *adapter*, *facade*, *bridge*, *flyweight*, *proxy*, *composite* and *decorator*. They are mostly expressed in UML class diagrams. Expressing most of them in HERM looks straightforward, however, the proxy, composite and decorator patterns feature non-trivial composition relations.

The *proxy* is a type whose data instances contain instances of another type ('real type') and both of them are generalized to a third type (the general type).

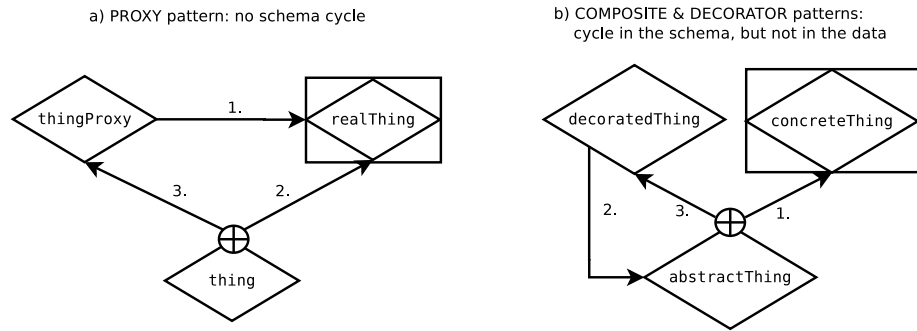


Fig. 1. Two non-trivial structural design patterns expressed in HERM.

A natural way of modeling it by HERM is using a cluster type whose subtypes are the proxy and the real type, and the former builds on the latter. Fig. 1/a) shows this setting. Numbers denote a possible order of schema construction. Although it is non-trivial, it is a well-formed schema in conventional HERM.

The *composite and decorator* patterns, however, have a self-referencing property, i.e. a data instance of a composite or decorated version of a type may contain another data instance(s) of the same type, which again may either be composites or decorated instances, or pure (concrete) instances. This can be expressed again by a cluster type, but here, a decorated or composite instance will depend not directly on a concrete instance but on a general (abstract) instance, which is the cluster type. This results in a directed cycle in the HERM schema structure, and therefore, cannot be considered as conventionally well-formed. See Fig. 1/b). However, there is no cycle in the data, if the composition has a proper layering on the instance level, for example, a concrete data instance becomes abstract so that it can be decorated, but a decorated instance is becoming abstract too, so that it can be further decorated. It does not mean actually a cycle but an iterative data construction.

2.4 Structural Recursion in Schemata

Structural recursion occurs wherever a type is constructed so that its data instances may be built on other instances of the same type. Examples of structurally recursive relationships - or relationships that can be modeled using structural recursion - include part-of or nested-into relationships such as departments of organizations, or a hierarchy of heterogeneous administrative regions (esp. if data is from multiple sources without a common prescribed layering - e.g. the disjoint union of regions of different countries on multiple administrative levels), or when data is organized into a linked list or other structure where the depth of the hierarchy is not bounded.

Note that structural recursion may not be the single valid way of modeling a phenomenon. It is especially useful if the nature of the domain is so that the composite objects have no subsistence on their own, but they are dependent on

other composite objects of the same type, which is in some cases even reflected in the identification (key attribute inheritance) as well. A region, for instance, does not have its global worldwide unique identifier on its own by nature, but its identification assumes the country of which it is part of, so its own identifier becomes a global key only with the inherited country identifier. A street in a city is in an even lower level, and inherits the country, region, city key attributes as part of its identification.

Another typical case of structural recursion is when the items are purely logical, they can be composed in various ways and their natural identification is based on the entire structure. An example is given in [11] where semi-automatic generation of trail signpost items is considered, where in the planning phase only the content is given, but is not assigned to any physical sign yet, so their whole semantical construct is inherently part of their natural key. A more common example is when formulas are the data instances and they are composed of atoms and operator symbols by structural recursion, and have no other natural key than their whole content with their structural composition. It can be specifically relevant to active databases with logical data implications [12]. Although a simplified derived representation of the key as identifier can be introduced into the schema later, this phenomena results in possibly complex keys when modeling the nature of the domain.

3 General Formulation and Proposed Extension

3.1 A general example of a structurally recursive schema

Let us consider the schema on Fig. 2 as a general pattern for unary structural recursion. This is a direct generalization of the logical trail signpost structure model presented in [11].

Type L is called the *initial type* (a.k.a. *leaf type* or *base type*), which can be either an entity (0th order) or a relationship (1st or higher order) type. We represent it as a relationship type which is the more general case. If it is an entity, its primary key is formed by one or more of its attributes. If it is a relationship type, it inherits the key attributes from the entity types it is based on, and the key is formed by these attributes, optionally extended by own attributes. The modeler may decide to override the inherited key attributes and compose a primary key for a relationship type based on one or more own attributes only.

The initial type L is generalized to a cluster type D , called the *domain union type* or *common type* which may add its own key or non-key attributes, entities or lower-order relationships it is based on, and may participate in relationships of higher order.

The structural recursion is realized by a relationship type R , called the *recursive type* (or *composite type*), each of whose data instances is based on either an instance of the initial type L or another instance of the composite type R . It can be modeled by a relationship type, which is based on D and is generalized by D at the same time.

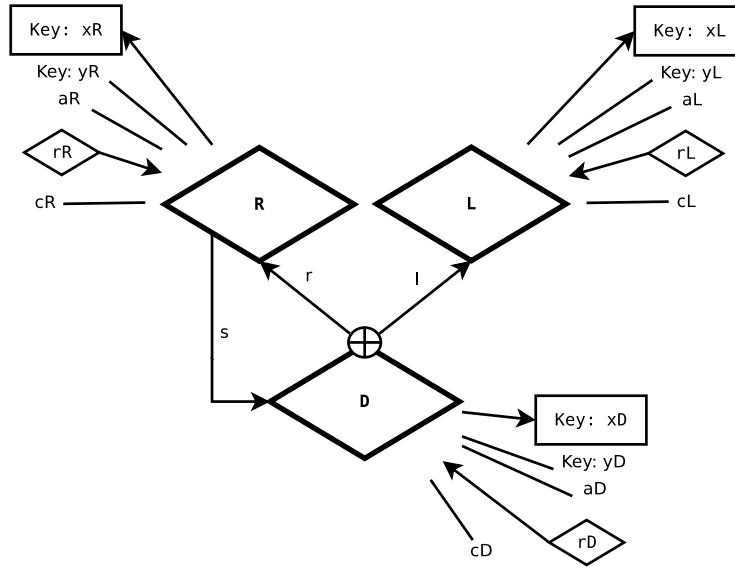


Fig. 2. General HERM schema pattern with unary structural recursion.

Although the model graph becomes cyclic, there should be no cycles in the data instances of any valid database instance. A special constraint is required to avoid cycles in the data: an assignment of a positive integer *rank* must be possible to each data instance r of R so that it must be higher than the rank of the data instance of R on which r is directly based (if r is not directly based on a data instance of L). Certainly, R may have its own attributes including key attributes, connected entities which it is based on and inherits key attributes from, as well as higher-order relationship types that are based on it.

Note that R inherits the key attribute(s) from D , and due to the structural recursion, this inheritance may be of an arbitrary depth, the length of the key of R (and D) may be variable, resulting a key with a complex domain, which follows the data instance composition by structural recursion. To avoid this, the schema designer can override the inherited key attributes by declaring an explicit primary key of R or D based only on its (local) attributes, but there are cases where it is not possible, and identification must be based on the entire sequence of the D -instances an instance of R is based on.

The notations of Fig. 2 are the following. Each element represents an attribute or relationship type set (i.e. stands for one or more items) with a specific characteristic to its respective type.

- xL, xR, xD denote inherited key attributes from entity types the respective relationship type is based on (xL is considered to be empty if L is an entity type);
- yL, yR, yD denote own attributes of the respective type which are part of the primary key (additionally to any inherited key attributes);

- aL, aR, aD are non-(primary-)key attributes of the respective type;
- rL, rR, rD are (higher-order) relationship types built on the respective types;
- cL, cR, cD are computed (derived) attributes of the respective types. We assume for the sake of simplicity that they denote single attributes, but the results can be trivially extended to a vector or derived attributes.

Note that $xR \cup xD \cup yR \cup yD$ must not be empty in order to have proper identification for the composite data instances of R .

3.2 Aggregation as an example of structurally recursive derived attributes

Aggregation may be defined for types R and/or D , resulting a derived attribute value based on the values taken from the structurally recursive composition of data instances. Similarly, any function or operation may be defined over the data instances of these types which take such values. In order not to lose the ability to reference any of these values in a translated schema in a way corresponding the composition structure, we must take extra care of keeping the translatability of such functions. We will thus define general aggregation functions for computing the derived attributes cL, cR, cD as an example and will consider their proper translatability.

To define a general aggregation, we assume two aggregative functions f_R and f_D defined for data instances of types R and D , which compute a value recursively, based on own attribute values of the respective instance i (and/or values taken from its connected relationships), and the respective aggregative function value of the instance on which i is directly based by the model structure. The zero-level aggregation is defined by a simple function f_L over data instances of L . Our aggregation functions F_L, F_R, F_D for each type and thus the values of the derived attributes cL, cR, cD are then being defined as follows:

- $\lambda.cL ::= F_L(\lambda) = f_L(\lambda.xL, \lambda.yL, \lambda.aL, \lambda.rL)$ if $\lambda \in L$,
- $\delta_l.cD ::= F_D(\delta_l) = f_D(F_L(\delta_l.l), \delta_l.xD, \delta_l.yD, \delta_l.aD, \delta_l.rD)$ if $\delta_l \in D$ a data instance based on L ,
- $\delta_r.cD ::= F_D(\delta_r) = f_D(F_R(\delta_r.r), \delta_r.xD, \delta_r.yD, \delta_r.aD, \delta_r.rD)$ if $\delta_r \in D$ a data instance based on R ,
- $\rho.cR ::= F_R(\rho) = f_R(F_D(\rho.s), \rho.xR, \rho.yR, \rho.aR, \rho.rR)$ if $\rho \in R$.

To define a unified aggregation function F for all the types, we may simply take their union:

$$F(\iota) = F_L(\iota) \text{ if } \iota \in L, F_D(\iota) \text{ if } \iota \in D \text{ and } F_R(\iota) \text{ if } \iota \in R.$$

Since the types are disjoint, and the above computation can be uniquely evaluated for each data instance composed by structural recursion, we can state the following:

Proposition 1. *The aggregate function F and the computation of the derived attributes cL, cR, cD are well-defined.*

4 Schema Translation to Non-Recursive Schemata

4.1 Translation Guidelines

Taking the above example as a basis, we want to give an equivalent alternative schema without the formal cycle of structural recursion, using the list type constructor [15] denoted by $[]$, as necessary.

Considering schema translation, we have identified the following guidelines as proposals for guiding automatic schema translation, keeping the interpretability and integrity of the original types as much as possible. The schema designer may choose and select some or all of them as relevant for an actual case, and this selection, acting as a directive, should determine the way of schema translation in an envisioned model-to-program framework. The resolution of the structurally recursive schema is considered to be intermediate schema, possibly ready for any further schema optimizations and (de)normalization procedures.

Information equivalence The translated schema must have the same expressivity as the original schema. Any valid database instance of one of the schemata must be mappable to the other schema so that one can tell for each type and attribute its correspondent(s) and they must preserve the structure of the information, i.e. the relational connections in the data. We can define an equivalence mapping (denoted by \equiv) from the original schema to the translated schema where each entity or relationship of the original schema with its attributes and connections has a structurally analogous counterpart in the translated schema, and vice versa. In most cases it will be trivial, and it will be shown for the non-trivial settings only. We will assume a valid database instance each time and will refer to its relations by the type names with capital letters as sets (relations), and their members by small greek letters as the data instances (tuples). The instance will not be named explicitly, and as the equivalence is shown for the generic case, it must be valid for all valid instances.

Self-containment principle During the schema translation, we want to preserve the compositionality of the schema for the aggregations. It means, each attribute value or connected relationship instance belonging to a data instance of a type, or is reachable from it along a directed path of the instance graph along the arrows defined by the schema, should remain reachable from the corresponding data instance in the translated schema, if it contributes to the specified aggregation computation of a derived attribute or it. For example, if an attribute in aL is referenced by the computation function of cR , then it must be reachable from the type in the translated schema to which cR belongs, using only the arrows in the HERM graph of the instance graph in terms of the translated schema. It also means, for example, that any of the attribute values of a data instance ρ of R contributing to the aggregation of cR of another data instance $\rho' \in R$ or of cD of a data instance $\delta \in D$ must remain reachable from ρ' or δ , respectively, if they are - even indirectly - based on ρ by structural recursion.

Inverse self-containment principle This states validity of the above self-containment principle for the inverse translation of the resulting schema to the original one. It means no attributes or any other parts of a type can be subordinated to a relationship in the translated schema which was not subordinated to its counterpart in the original schema. Subordinated means it is put 'under' the relationship, so that the relationship is directly or indirectly built on top of it.

Natural key principle Closely related to the self-containment principle, during the schema translation, we want to preserve the key composition of the types by the standard inheritance of HERM, introducing possibly complex key structures such as by list type construction so to give natural self-identification to the data instances. It is then up to the schema designer or database implementer at a later phase if (s)he wants to introduce surrogate keys inside the database schema for effectivity. However, these surrogate keys will not be visible to the users normally and they will still need a natural unique key for each type to be able to identify its data instances, and this should be based on the natural key set in the conceptual schema.

Duplication avoidance Another principle is we do not want to duplicate attributes or relationship types in the schema. For instance, if a cluster type has to be split so that some of its attributes will become part of another relationship type, there must be an exact unique location in the schema where such an attribute is connected, without declaring the attribute twice, for two different types.

Mixture avoidance Moreover, we want to keep the integrity and clear separation of the types in a way that although a type may be split into two or more types, but we do not want to mix attributes of formerly different types into one type in the translated schema.

Semantic unit encapsulation Lastly, by the principle of semantic unit encapsulation we try to avoid repetitive reference patterns, which means, if two or more relationship types is connected to two or more other relationship types in a similar way, and for the similar reason, because the combination of these other types form a semantic unit, than we make this semantic unit explicit by forming a relationship type which connects them, and the multiple relationship types which referenced them in a similar way will reference only this new, "encapsulated" relationship type. It helps to define a clearer, more understandable and overseeable schema, takes advantage of the structural expressivity of HERM and eliminates some arrow crossings.

Depending on these guidelines, we will give properly translated variants of the structurally recursive schema of Fig. 2 in the following. Information equivalence will be shown or explained where non-trivial. Self-containment and its inverse will be partially relaxed (Sect. 4.2) as well as semantic unit encapsulation (Sect. 4.5). Natural key, duplication and mixture avoidance will be kept throughout all given translation patterns.

4.2 Translation by Introducing a Relationship Type

Introducing an extra relation in the conceptual schema instead of the connection denoted by role s seems to be the most straightforward resolution as it is used in classical examples ([15] and others). However, it is worth to note that when coming to the logical schema obtained by classical schema implementation, it is likely to be eliminated and transformed into a foreign key in the table of R , referencing D , which looks visually closer to the original schema with structural recursion.

Fig. 3 shows variants of this method (attributes and further relationship connections omitted). Depending on the role and content of D , it may be included or omitted, or even left out from the whole schema (if D has substantial content, also contributing to the aggregation, then variant a) is to be chosen). Choosing any of these patterns, however, breaks the self-containment principle for R , since a data instance of R will structurally not 'contain' (i.e. entail or reference by itself) those D -instances on which it is built by structural recursion, and are therefore integral parts of it. Any aggregation along the structurally recursive construct must be defined through S , which is a higher-order type, formally became external to R .

Considering key inheritance paths, the potential problem becomes more obvious: if identification (i.e. the key) of R is based on the data instances involved in its structurally recursive composition, then the desired key inheritance is not achieved by any of these schemata. Relation S (or SL, SR) inherits the key attributes from D and R , but R does not inherit any key through it. In variant b), R inherits the key of L , with its initial component instance, but does not inherit any key from other R data instances. Therefore, it can be used when this is allowed. The other variants a), c) and d) do not even inherit the key of L towards R .

Information-equivalence of these schemata with the original one is obvious, if data acyclicity can be ensured w.r.t. S (or SR). If the self-containment guideline is not strictly applied and key inheritance in R is overridden, these are simple options to choose from.

As an example, the information equivalence mapping of R for variant a) is:
 $R_{orig.s} \equiv R.(S.r)^{-1}.d$, or simply expressed by the relation S .

Similar equivalences can be given for the other variants, while equivalence for all other types and attributes is the identity function.

However, complex integrity constraints are potentially required for ensuring the acyclicity of the data [15], as the transitive closure of S (or SR) must be irreflexive.

If identification of an R -instance is inherently based on the data instances appearing in its structurally recursive composition, then we must seek a different solution.

4.3 Translation by Splitting the Composite Type

To go beyond the above solutions, when keeping the self-containment of type R is necessary, especially if identification is based on the structurally recursive

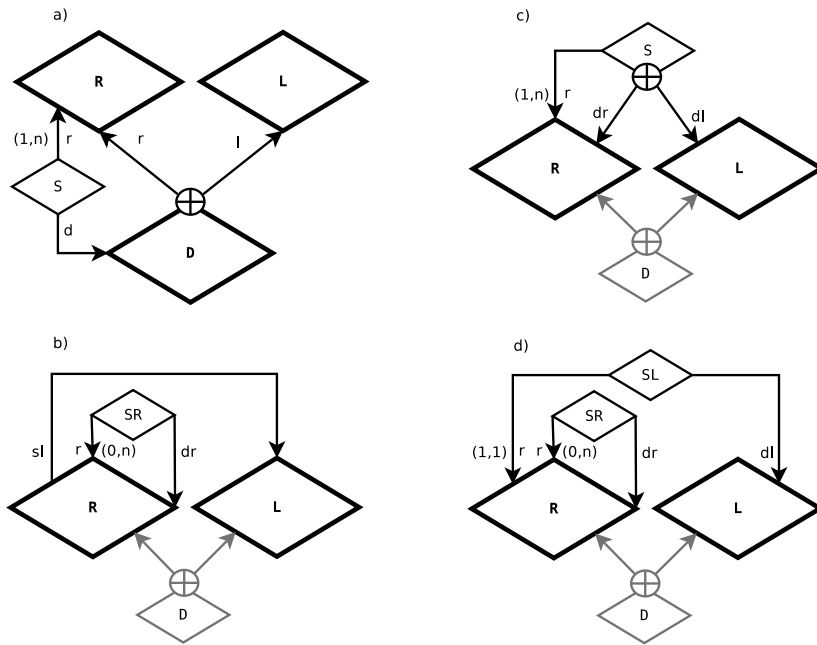


Fig. 3. Translation of the recursive model: adding a relationship type.

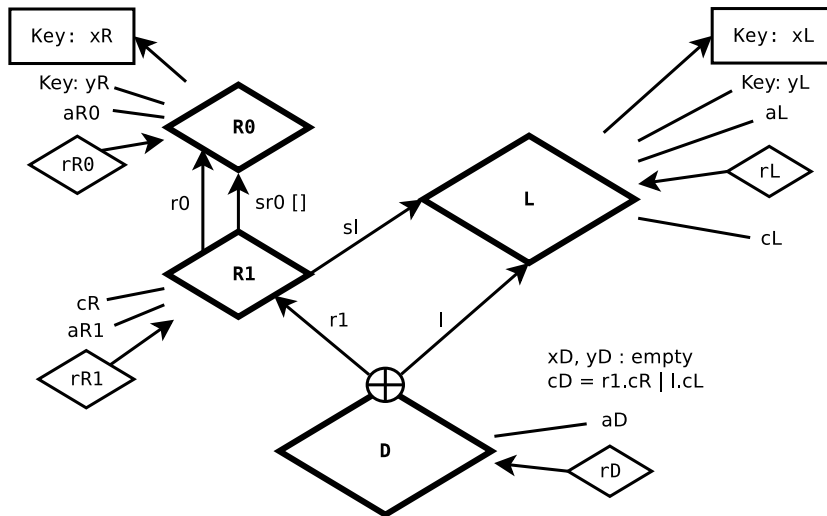


Fig. 4. Translation of the recursive model: mid complexity case.

construct, and the aggregation of cR must be kept inside type R , we discuss two cases, a simpler and a more general.

For the simpler case, we assume D has no own key attributes, its own attributes or connected relationships (if there is any) do not contribute to the aggregation of cR , and has no specific aggregation function (cD is either nonexistent or is equal to cR or cL , depending on the actual instance of the cluster type). Putting it to simple words, D may have generally no effect on the aggregations and the structural recursive composition, it acts only as a generalizer façade over L and R and may add some simple higher-level, non-aggregative attributes and may participate in higher-order relationships on behalf of the whole structurally recursive part of the schema.

We have to consider possible attributes and connected relationships of R , which of them do contribute to the aggregation of cR and which do not. We denote by $aR0$ those attributes of the attribute set aR which contribute to the aggregation, and by $aR1$ those which do not contribute to the aggregation. We use a similar notation $rR0$ and $rR1$ for the relationships rR as well.

To eliminate the formal cycle in the schema graph, we use the list (or array) constructor, which means a relationship type can be defined so that its component type has multiple (zero or more) data instances assigned to the same instance of that component type, in an ordered (indexed) way. This is denoted by a $[]$ in the HERM schema graph.

Anything that contributes to the aggregation must be defined as part of a base type, of which a list is formed and another, higher-order relationship type is defined over it, where the actual aggregation is taking place. This implies the type R to be split into two relationship types $R0$ and $R1$, where $R1$ contains cR and everything else not contributing to the aggregation, and builds on a list of $R0$, which then contains the original key of R and everything contributing to the aggregation. $R0$ must also build on the initial type L of the structural recursion, since every data instance of R references a data instance of L at the end. For each instance of $R1$ we will have a specific instance of $R0$ which contains the rest of its own attributes, and zero or more other instances of $R0$ in an order corresponding the structurally recursive composition of "previous" instances of R on which the original instance R is built on. Although this can be merged into one list with an obligatory first instance, it is more expressive to model it separately, so $R1$ will have a duplicate (a non-list and a list) schema connection to $R0$.

The resulting schema can be seen on Fig. 4.

Proposition 2. *The schema on Fig. 4 is information-equivalent to the schema on Fig. 2 given the conditions of the first paragraph in this subsection, with obeying the guidelines of Sect. 4.1, with $R1$ representing type R of the original schema, with the natural key attribute inheritance implied by the schema (including key attributes from $R1.r0$ as $R1.r0.xR, R1.r0.yR$, each key attribute from each list member of $R1.sr0$ in its specific order $R1.sr0[i].xR, R1.sr0[i].yR$, and the key attributes from $R1.sl$ as $R1.sl.xL, R1.sl.yL$), with the added inclu-*

sion constraint $R1.sl \subseteq D.l$, and with the following iterative aggregate definition (where $\rho \in R$, $n \in \{0, 1, 2, \dots\}$ is the length of the list $\rho.sr0$):

- $F_{R1}(\rho) = f_R(F_{RL}^n(\rho), \rho.r0.xR, \rho.r0.yR, \rho.r0.aR0, \rho.r0.rR0)$;
- $F_{RL}^i(\rho) = f_R(F_{RL}^{i-1}(\rho), \rho.sr0[i].xR, \rho.sr0[i].yR, \rho.sr0[i].aR0, \rho.sr0[i].rR0)$ for each $i \in \{1, \dots, n\}$;
- $F_{RL}^0(\rho) = f_L(\rho.sl.xL, \rho.sl.yL, \rho.sl.aL, \rho.sl.rL)[= \rho.sl.cL]$.

Information-equivalence is achieved through the following mappings:

- For type L and D : identity in both directions (see conditions for D).
- Type R is mapped to $R1$ with correspondances
 - $\Pi_{aR1,rR1,cR}(R_{orig}) \equiv R1$
 - $\Pi_{xR,yR,aR0,rR0}(R_{orig}) \equiv R1.r0$
 - $R_{orig}.sl \equiv R1.sl$ and $R.s \equiv R1.sl.(D.l)^{-1}$ if $R.s$ is an L -based D -instance
 - $R_{orig}.sr \equiv R1.sr0[1]$ and $R.s \equiv R1.sr0[1].(D.r1.r0)^{-1}$ if $R.s$ is an R -based D -instance
 - Furthermore, $(R1.sr0[], R1.sl)$ is generally equivalent with the sequence of the structurally recursive composition of the corresponding R -instance in the original schema, with the R -instances projected as $\Pi_{xR,yR,aR0,rR0}R$.

Remark. The recursive definition of F_{R1} in the translated schema is usually rewritable to closed form, making the aggregation more effective. This is beneficial (if the array is stored compactly), since the schema translation implies multiple computations of the same function values, due to multiple references of $R0$ data instances or sequences in $R1$ data instances.

4.4 Translation by Splitting Both the Composite and the Domain Types

In the most general case, when type D has its own key attributes, aggregate function(s) for cD and/or its attributes or relationships contribute to the aggregation of cR or cD , the situation becomes far more non-trivial. This is due to the effect that an arbitrary data instance of D or R is built on other instances of D and R (and eventually, an instance of L) by structural recursion, and the key attributes must be properly reachable - along a directed path in the structure - and inherited for each data instance, as well as the other aggregation-contributing attributes must be reachable and correctly referenced.

Besides keeping the separation of type R into $R0, R1$ as in the previous section, we will have to use the same separation technique for D to organize key and aggregation-contributing attributes (and possible aggregation-contributing relationships) into a base type $D0$, and keep the rest of the attributes and relationships together with the aggregate attribute cD in a type $D1$ which will build on $D0$ and substitute the original type D .

But where to put type $D0$ in the translated schema, so that the reachabilities are ensured and the identification as well as the aggregation will not break, and the general guidelines (see Sect. 4.1) set for the translation are obeyed?

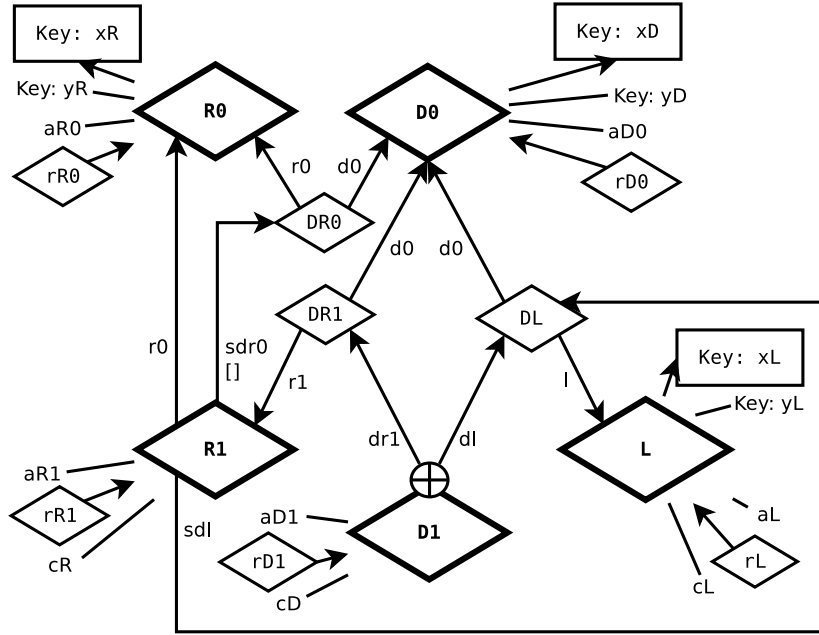


Fig. 5. Translation of the recursive model: complex, general case.

After several considerations and wrong tries, one will likely get a translated schema similar to what is depicted on Fig. 5. The reasoning below makes the following statement clear:

Proposition 3. *The schema on Fig. 5 is information-equivalent to the schema on Fig. 2, given the general guidelines of schema translation as introduced in Sect. 4.1 and the following conditions:*

- Attribute sets aR and aD are decomposed to $aR0 \cup aR1$ and $aD0 \cup aD1$, respectively, where $aR0$ and $aD0$ contribute to the aggregations cR and cD .
- Similarly, relationship sets rR and rD are decomposed to $rR0 \cup rR1$ and $rD0 \cup rD1$, respectively, where $rR0$ and $rD0$ contribute to the aggregations cR and cD .
- $D1$ represents original type D , $R1$ represents original type R in the translated schema.
- Identification is done by the natural key attribute inheritance implied by the schema (including key attributes from $R1.r0$ as $R1.r0.xR, R1.r0.yR$, each key attribute from each list member of $R1.sr0$ in its specific order $R1.sr0[i].xR, R1.sr0[i].yR$, and the key attributes from $R1.sl$ as $R1.sl.xL, R1.sl.yL$).
- The following integrity constraints must hold:
 - $D0 \subseteq DR1 \cup DL \subseteq D1$
 - $DR0 \subseteq \Pi_{r1.r0,d0}(DR1 \bowtie R1)$
 - $D1 \subseteq DR1 \cup DL$

- The following iterative aggregate definition is given which is equivalent to F in original schema (where $n \in \{0, 1, 2, \dots\}$ is the length of the list $\rho.sdr0$ for $\rho \in R1$):
 - $\lambda.cL ::= F_L(\lambda) = f_L(\lambda.xL, \lambda.yL, \lambda.aL, \lambda.rL)$ if $\lambda \in L$ (as in the original schema),
 - $\delta_l.cD ::= F_{D1}(\delta_l) = f_D(F_L(\delta_l.dl.l), \delta_l.dl.d0.xD, \delta_l.dl.d0.yD, \delta_l.dl.d0.aD, \delta_l.dl.d0.rD)$ if $\delta_l \in D1$ a data instance based on DL ,
 - $\delta_r.cD ::= F_{D1}(\delta_r) = f_D(F_{R1}(\delta_r.dr1.r1), \delta_r.dr1.d0.xD, \delta_r.dr1.d0.yD, \delta_r.dr1.d0.aD, \delta_r.dr1.d0.rD)$ if $\delta_r \in D1$ a data instance based on $DR1$,
 - $\rho.cR ::= F_{R1}(\rho) = f_R(F_{DRL}^b(\rho), \rho.r0.xR, \rho.r0.yR, \rho.r0.aR0, \rho.r0.rR0)$;
 - $F_{DRL}^i(\rho) = f_D(F_{DRL}^i(\rho), \rho.sdr0[i].d0.xD, \rho.sdr0[i].d0.yD, \rho.sdr0[i].d0.aD0, \rho.sdr0[i].d0.rD0)$ for each $i \in \{1, \dots, n\}$;
 - $F_{RDL}^i(\rho) = f_R(F_{RDL}^{i-1}(\rho), \rho.sdr0[i].r0.xR, \rho.sdr0[i].r0.yR, \rho.sdr0[i].r0.aR0, \rho.sdr0[i].r0.rR0)$ for each $i \in \{1, \dots, n\}$;
 - $F_{DRL}^0(\rho) = f_{D1}(\rho.sdl)[= \rho.sdl.l.cL]$.

Information equivalence can be constructed in a similar way as in the previous section, extending it for type D analogously to type R .

Explanation. In order to make any aggregation (iterative computation) possible, under the reachability condition, having an acyclic schema, we have to separate the types with aggregation into (at least) two types, where one is based on a list of the other (directly or indirectly). It must be applied to R and D . Type L does not have to be decomposed. However, the higher-order part of R with the aggregated attributes and those not participating in the aggregation must reference the base type part of D with its attributes participating in the aggregation cR so that the aggregation can take all the relevant attribute values along the structural recursive composition of any data instance r . Therefore, the type $R1$ which becomes the correspondent of R in the translated schema must build on $R0$ and $D0$, and L somehow directly or indirectly.

Since attributes are not allowed to be merged from different types of the original schema to the translated schema, these separations must be disjoint, resulting in 5 different types: $L, R0, R1, D0, D1$. Based on which attributes or relationships are keys or contribute to the aggregation, the separation of these among the above types becomes straightforward.

In order to have proper identification (making a variable-length keys in this case for $R1, D1$), key attributes must be put into the base types $R0, D0$, so that the primary key of $R1, D1$ (respectively) can be composed similarly to the aggregation, as a list of the inherited key attributes. The key of $D0$ is inherited by $R1$ only for the structurally recursive part, not for the data instance $\delta \in D$ for which $\delta.r = \rho$.

Consider the initial step of the structural recursion, given a data instance $\lambda \in L$. If it takes part of any structural recursion, it has to be referenced by an instance $\delta_l \in D$, so that $\delta_l.l = \lambda$ in the original schema. If an instance $\rho \in R$ is based on this, then $\rho.s = \delta_l$ so that not only the own attributes of L must be reachable in the instance graph of the translated schema along

the structural arrows, but also all attributes of D which do contribute to the aggregation. Therefore, $R1$ has to be based on - directly or indirectly - L and $D0$ (additionally to $\lambda \in L$, there must be a data instance $\delta_0 \in D0$ which is referenced by $\rho_1 \in R1$, the translated correspondent of ρ). On the other hand, because δ_l is a full-fledged data instance of D in the original schema, and has an aggregated value cD , it must correspond to an instance $\delta_1 \in D1$ in the translated schema with all the other attributes of D not contributing to the aggregation. So the pair (λ, δ_0) becomes a semantic unit referenced by both $R1$ and $D1$, and therefore, it is modeled as a new relationship type DL by the semantic encapsulation principle, representing the "internal" manifestation of the type of L -based D -instances (which participates in the structural recursion), while the "external" manifestation of the cluster type D becomes the cluster type $D1$.

Analogously, $R0, D0$ can be observed a semantic unit as the internal manifestation of R -based D -instances contributing to the composition of a data instance $\rho \in R$ by structural recursion. It becomes the intermediate type $DR0$ which is referenced as the list of translated R -based D -instances building up ρ by structural recursion (role $sdr0[]$). Although not referenced more than once in the schema, its convenience is being the list member type (otherwise we would need two separate array-type references). At the depth of the recursive composition path stands an L -based D -instance in the original schema, which is mapped to a DL -instance in the translated schema and is referenced by $R1$ (role sdl). The translation of the role s with its all iterations through roles $D.r$ and $D.l$ in the original schema becomes the role pair $sdr0[], sdl$. The "own" attributes and relationships of ρ contributing to the aggregation are referenced directly as another role $r0$ towards the type $R0$.

The intermediate types $DR0$ and DL and their references make clear the different roles of referenced $D0$ -instances from the various directions (whether it is a translation of an L -based or an R -based D -instance and on which level in the structural recursion from the point of a translation of a particular R -instance).

The internal manifestation of the R -based part of cluster type D in the translated schema is the $DR1, DR0$ type pair, and at the same time the external manifestation of the original type R becomes $R1$. The external manifestation of the full original cluster type D becomes $D1$ which is composed of DL and $DR1$, adding its own aggregate attribute cD and the other related items not contributing to the aggregation.

Therefore, the required attributes and relationships can be properly reached in the translated schema on Fig. 5 for the aggregation of cR from type $R1$, and the same is true for cD from $D1$. At the same time, all the guidelines of Sect. 4.1 hold.

Furthermore, Fig. 6 shows a refinement of this schema with the relaxation of the guidelines, by abolishing the inverse self-containment principle. R must be fully covered by D , so parts of D (here, $D0$) can be subordinated to $R1$. If this constraint holds in the original schema, this translation pattern can also be applied.

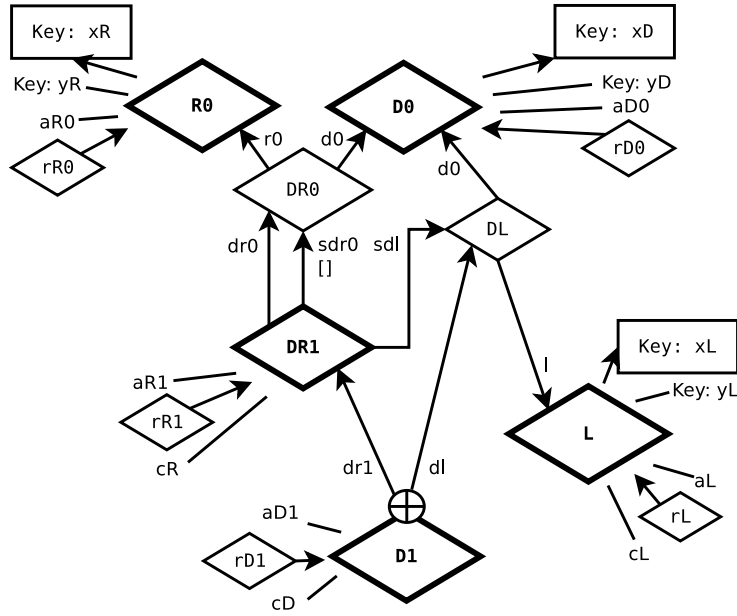


Fig. 6. Refined translation of the recursive model in the complex, general case, when $D0$ is subordinated to R .

4.5 Flattening the Split Schemata

If the guideline of semantic unit encapsulation is not in effect, the intermediate relationship types DL and $DR1$ can be eliminated from the schemata of Fig. 5 and Fig. 6 by linking the higher-order relationships built on them directly to the entities or relationships they connect. This results the 'flattened' schemata depicted on Fig. 7 and Fig. 8, respectively. For the latter case, a full covering of both L and R is assumed by D (so each data instance of L and R must participate in D). This constraint must hold in the original schema for information equivalence. The main difference is the reachability of $D0$ via $R0$ and L , whether $D0$ is part of the composition of the latter relationships or only directly of $D0$. The latter case assumes the inverse self-containment guideline is not in effect.

The information equivalence of these schemata under the above conditions is obvious, and similar propositions may be formulated for these flattened schemata as for the ones in the previous section.

Note that we have kept the other principles (mixture or duplication avoidance, natural key principle) strictly in effect for all of the translation patterns. If one or more of these are put aside, new patterns can be constructed. However, a reasonable method is to keep these principles for the resolution of the structural recursion, and relax them only if the schema needs further optimization. This way, a clear separation is achieved in the method: we keep the resolution of the structurally recursive schema separated from further schema optimizations.

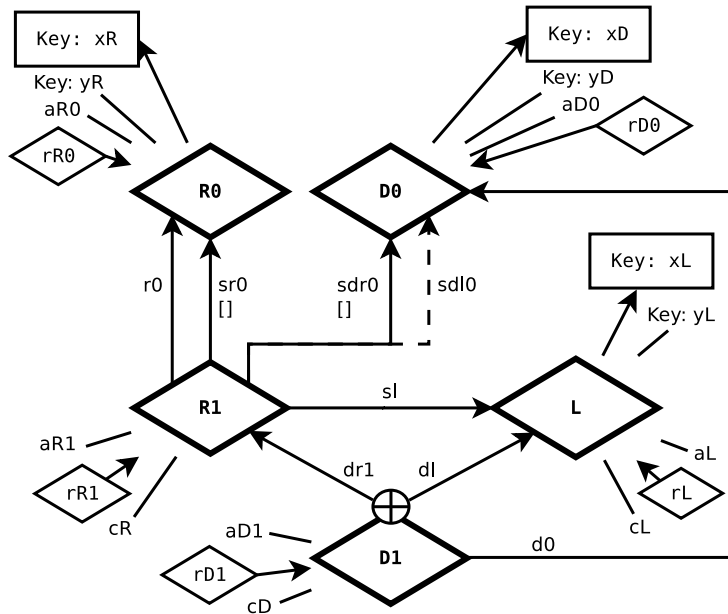


Fig. 7. Translation of the recursive model: complex, general case - flat variant (note: *sd0* is optional, as it can be merged with *sdr0* to form a common *sd0*).

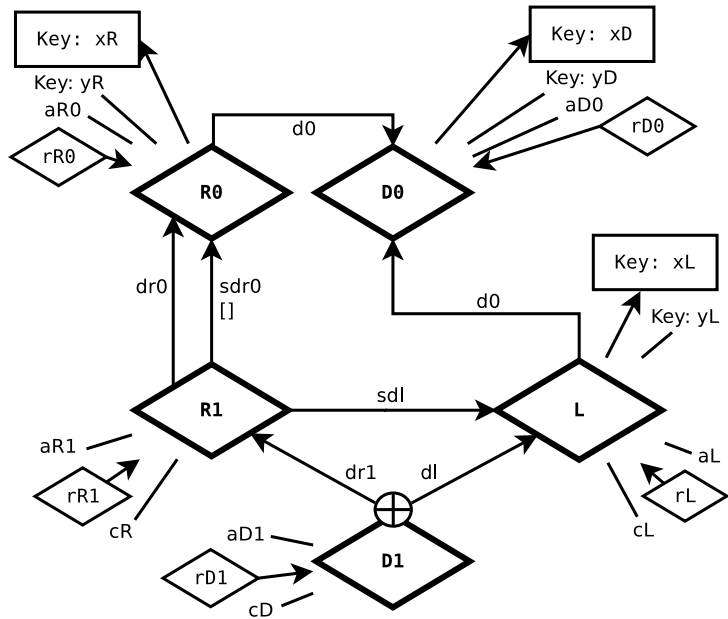


Fig. 8. Flattened translation of the recursive model in the complex, general case, when D is covering for both L and R.

5 General Rule Formulation

Depending on the actual setting, one or more of the above translation patterns can be applied. We can summarize all these options with their conditions in the following algorithm:

- Does the key sequence implied by the structural recursion via D have to be inherited via s to R and is the self-containment guideline in effect?
 - If not then choose a schema from Fig. 3/a), c) or d) depending on the role of D (based on its content and if its self-containment is relevant w.r.t. S) and formulate an own key for R (overriding key inheritance in the original schema).
 - Otherwise, identification of R must depends on its structurally recursive composition (partly or fully). Continue with the next step.
- Is the key of L the only key implied by the structural recursion via D contributing to the identification of R (i.e. a data instance of R does not inherit any key attributes from the sequence of R -instances it is built on by structural recursion) and is the self-containment guideline not in effect?
 - If yes then choose schema on Fig. 3/b) and make the key of L part of the key of R .
 - Otherwise, identification of R depends on the full key chain of its structurally recursive composition. Continue with the next step.
- Are the restrictive conditions in Sect. 4.3 hold for D ?
 - If yes then choose schema on Fig. 4, otherwise continue with the next step.
- Is D covering for both R and L (i.e. each data instance of R and L must participate in D) and can the inverse self-containment guideline be put aside?
 - If yes then you may chose schema on Fig. 8, otherwise continue with the next step.
- Is D covering for R (i.e. each data instance of R must participate in D) and can the inverse self-containment guideline be put aside?
 - If yes then you may choose schema on Fig. 6, otherwise continue with the next step.
- If none of the following conditions hold, or the offered schema is otherwise inconvenient, the schema of the most general case as on Fig. 5 or its flattened variant on Fig. 7 must be taken, based on whether the semantic unit encapsulation guideline is in effect.
- The schema is well-formed without a formal directed cycle in its structure. Considering it as an intermediate schema, make further optimizations on the schema as usual, based on the modeling methodology on conventional schemata being applied. This should be guided by the other translation principles in effect (mixture or duplication avoidance, natural key principle, or even the information equivalence may be relaxed in specific cases).

6 Conclusion and Future Work

In this paper, we have shown how the Higher-Order Entity-Relationship model (HERM) can be extended to allow unary structural recursion to be expressed, which formally contains a cycle in the model graph (thus not considered well-formed as a conventional schema), but without actual cycles on the instance (data) level. Several variants of reasonable translation patterns have been developed and presented in order to achieve an equivalent, conventionally well-formed schema. An algorithm-like rule summarises their applicability, which is based on the specific properties of the original schema, as well as some general guidelines determining the actual methodology of schema translation. The result is an intermediate schema possibly involving complex key domains by list type construction, and is ready for further optimization.

Future issues and open questions include further generalization towards more complex structures, investigation on the effect on database constraints and constraint management of structurally recursive schemata, and further discussion on the translation variants, for example, regarding usage patterns and query efficiency.

Considering the modeling-to-programming initiative, this case study gives an example of a relatively simple-looking but complex scenario, where model translation is achieved by choosing from pre-designed schema patterns, and the choice is determined by explicit guidelines and inherent model properties as well. It can be directly built into a model-to-schema translator engine or similar software tool, and used as a scenario for more considerations related to the nature of model translation and structural recursion. A more general contribution of this paper is the collection of translation guidelines which can be used as possible directives set on or off by the modeler or schema designer for model translations.

References

1. AlBdaiwi, B., Noack, R., Thalheim, B.: Pattern-based conceptual data modelling. *Frontiers in Artificial Intelligence and Applications* **272**, 1–20 (01 2014). <https://doi.org/10.3233/978-1-61499-472-5-1>
2. Arenas, M., Libkin, L.: An information-theoretic approach to normal forms for relational and xml data. *Journal of the ACM (JACM)* **52**(2), 246–283 (2005)
3. Bahmani, A., Naghibzadeh, M., Bahmani, B.: Automatic database normalization and primary key generation. In: 2008 Canadian Conference on Electrical and Computer Engineering. pp. 000011 – 000016 (06 2008). <https://doi.org/10.1109/CCECE.2008.4564486>
4. Chen, P.: Entity-relationship modeling: Historical events, future trends, and lessons learned. In: *Software pioneers*. pp. 296–310. Springer (2002)
5. Codd, E.F.: A relational model of data for large shared data banks. *Commun. ACM* **13**(6), 377387 (Jun 1970). <https://doi.org/10.1145/362384.362685>, <https://doi.org/10.1145/362384.362685>
6. Date, C.: *Logic and Databases: The Roots of Relational Theory*. Trafford Publishing (2007), <https://books.google.de/books?id=2egNzTk871wC>

7. Embley, D., Thalheim, B.: Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges. Springer Berlin Heidelberg (2012), <https://books.google.de/books?id=oWmp10vBI7cC>
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1 edn. (1994)
9. Klettke, M., Thalheim, B.: Evolution and migration of information systems. In: The Handbook of Conceptual Modeling: Its Usage and Its Challenges, chap. 12, pp. 381–420. Springer, Berlin (2011)
10. Krtzsch, M., Hitzler, P., Zhang, G.: Morphisms in context (2005), www.aifb.uni-karlsruhe.de/WBS/phi/pub/KHZ05tr.pdf.
11. Molnr, A.J.: Conceptual modeling of hiking trail networks with logical rules for consistent signage planning and management. In: Thalheim, B., Tropmann-Frick, M., Jaakkola, H., Kiyoki, Y. (eds.) Proceedings of the International Conference on Information Modelling and Knowledge Bases (EJC 2020). KCSS, vol. 2020/1, pp. 1–25. Department of Computer Science, Faculty of Engineering, Kiel University (June 2020). <https://doi.org/10.21941/kcss/2020/1>
12. Paton, N.W., Díaz, O.: Active database systems. ACM Comput. Surv. **31**(1), 63–103 (Mar 1999)
13. Thalheim, B.: The conceptual model = an adequate and dependable artifact enhanced by concepts. In: Information Modelling and Knowledge Bases, volume XXV of Frontiers in Artificial Intelligence and Applications, 260. p. 241254. IOS Press (2014)
14. Thalheim, B.: Foundations of entity-relationship modeling. Annals of Mathematics and Artificial Intelligence **7**, 197–256 (03 1993). <https://doi.org/10.1007/BF01556354>
15. Thalheim, B.: Entity-Relationship Modeling: Foundations of Database Technology. Springer (2000)
16. Thalheim, B., Jaakkola, H.: Model-based fifth generation programming. In: Dahanayake, A., Huiskonen, J., Kiyoki, Y., Thalheim, B., Jaakkola, H., Yoshida, N. (eds.) Information Modelling and Knowledge Bases XXXI - Proceedings of the 29th International Conference on Information Modelling and Knowledge Bases, EJC 2019, Lappeenranta, Finland, June 3-7 2019. Frontiers in Artificial Intelligence and Applications, vol. 321, pp. 381–400. IOS Press (2019). <https://doi.org/10.3233/FAIA200026>, <https://doi.org/10.3233/FAIA200026>