Mapping between Alloy specifications and database implementations

Alcino Cunha
DI-CCTC
Universidade do Minho
Braga, Portugal
Email: alcino@di.uminho.pt

Hugo Pacheco
DI-CCTC
Universidade do Minho
Braga, Portugal
Email: hpacheco@di.uminho.pt

***Abstract*—The emergence of lightweight formal methods tools such as Alloy improves the software design process, by encouraging developers to model and verify their systems before engaging in hideous implementation details. However, an abstract Alloy specification is far from an actual implementation, and manually refining the former into the latter is unfortunately a non-trivial task. This paper identifies a subset of the Alloy language that is equivalent to a relational database schema with the most conventional integrity constraints, namely functional and inclusion dependencies. This semantic correspondence enables both the automatic translation of Alloy specifications into relational database schemas and the reengineering of legacy databases into Alloy. The paper also discusses how to derive an object-oriented application layer to serve as interface to the underlying database.**

I. Introduction

Model driven software engineering (MDSE) aims at improving the software development process by making extensive use of models at different levels of abstraction. In this model-centric approach, implementations are obtained by step-wise refinements of high-level specifications, ideally deployed as automated models transformations. Formal methods (FM) can play a key role on MDSE, not only on the model verification task, but also in establishing the correctness of model transformations. However, they tend to be neglected in common (non-critical) software development scenarios due to the high costs involved.

Alloy [1] is an increasingly popular modeling language that is particularly well-suited to serve as the ”Trojan horse” of FM in the overall MDSE community. It embodies the so-called lightweight approach to formal methods [2]: a language based on simple mathematical concepts, sharing some resemblances with typical object modeling languages such as UML, but combined with a powerful automatic SAT analyzer that brings the power of formal verification to the average software engineer.

Although Alloy is reaching a mature state as a modeling language, considerable work is still needed to unleash its full potential for MDSE. In particular, more theory and tool support is needed to help bridge the gap between specifications and implementations. In this paper we attempt precisely to establish the foundations of a framework to map between Alloy specifications and database-intensive applications. As

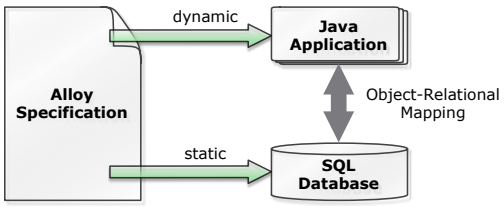


Fig. 1. Mapping Alloy specifications to database-intensive applications.

seen in Figure 1, the basic idea is to derive from the same specification the following artifacts:

- A relational database schema where the data model will be materialized. Whenever possible, the static constraints prescribed in the specification should be enforced by the database schema in order to avoid integrity problems.
- An object-oriented application layer to serve as interface to the programmer. All dynamic aspects of the specification, namely operations, will also be ported into this layer.
- An object-relational mapping (ORM) to handle the persistence of objects in the relational database.

In this paper we will mainly focus on the first artifact, i.e, the connection between Alloy specifications and relational database schemas.

After briefly presenting the Alloy modeling language (Section II) and recalling a formal definition of database schemas (Section III), we identify a subset of Alloy whose expressive power is similar to that of database schemas enriched with functional and inclusion dependencies (Section IV). We then formalize this correspondence by providing forward and reverse mappings between this Alloy subset and database schemas (Section V). They are also shown to be correct in respect to a database equivalence notion that ignores redundant attributes. These bidirectional mappings lay the foundation for not only migrating Alloy specifications to relational database implementations, but also reverse engineering abstract models from legacy databases. We then present the first steps toward the generation of the object-oriented application layer (Section VI), and discuss some technical details of the implemented prototype (Section VII). After presenting some related

```
sig Name {}
abstract sig Obj {
  name : one Name
}
fact {
  all n : Name | lone name·n
}
sig Dir, File extends Obj {}
sig FS {
  objects : set Obj,
  root : one (Dir & objects),
  parent : objects → lone (Dir & objects)
}
assert {
  all fs : FS | some fs·objects
}
pred cd [f, f':FS, d:Dir] { ... }
pred mv [f, f':FS, o:Object, d:Dir] { ... }
```

Fig. 2. Alloy specification of a filesystem.

work (Section VIII), the paper concludes with a synthesis of the main contributions, along with ideas for future work (Section IX).

II. The Alloy Language

In this section, we illustrate the Alloy language with a popular modeling example of a file system adapted from [1]. The model, shown in Figure 2, consists of a series of paragraphs, each specifying a signature, a fact, a predicate, or an assertion. Signatures realize the notion of type or class in Alloy and denote sets of atoms. Similarly to object-oriented languages, it is possible to specify an hierarchy between signatures (using the keyword **extends**), and to declare some signatures abstract. In this particular case, objects of a file system are either files or directories.

A signature declaration may also comprise a set of fields, that model relationships between signatures. For example, *name* associates objects with their names. In this field declaration, **one** is a multiplicity constraint stating that each object has exactly one name. The remaining fields state that a filesystem has a set of objects and one root directory that is an object of the filesystem, and that all the objects in a filesystem may have one parent directory that, again, is an object of the filesystem.

At this point, it is noticeable that, despite the natural and minimalist language, it is possible to express directly in field declarations intricate relationships between the entities of the data model. However, not all invariants can be expressed in this way, and more elaborate constraints can be defined as explicit facts, that are essentially first-order logic formulas that must always hold. In this particular case, we have a fact stating that $name$ must be injective, i.e, all objects have different names. Notice how the relational dot join operator (relational composition) is used to obtain all objects associated with a given name n. This is possible because all entities in an Alloy model are relations, including signatures, that are unary relations, and scalars, that are unary relations with a single tuple.

Analogously to functions in regular programming languages, predicates define parameterized formulas that can be reused several times in a specification. In Alloy they are typically used to model operations - since instances are immutable, these are specified by stating the relation between pre- and post-states. As a convention, post-states are usually primed in the list of arguments. Here we have two operations: *cd* and *mv* denote the change directory and move object operations, respectively. Their specification is not given, because in this paper we focus only the static aspects of a model.

Assertions specify formulas that should hold as a consequence of the model's facts. They can be used to check for model consistency, or to check the correctness of operations. The Alloy analyzer checks assertions by feeding them into an off-the-shelf SAT solver. A counter-example is shown when an assertion is not valid. In our example, the assertion states that no filesystem can be empty. This holds because there must always exist a root which is also an object of the filesystem. In fact, we could have used the multiplicity **some** instead of **set** in the declaration of the field *objects*. Assertions only make sense at the specification level, and have no effect on the final implementation. As such, they will be ignored in the remaining of the paper.

III. Database Schemas

In the traditional relational model, relations are unordered and tuples are partial functions from attributes to atoms, where the attributes of each relation must have distinct names. In Alloy, a relation $R \subseteq S_1 \times \ldots \times S_{|R|}$ is a subset of an ordered cartesian product of signatures. Notice that there may be repeated signatures in a relation declaration: signatures just play the role of types and cannot be used as attribute names. To better match this notion of relation, we will specify database schemas using ordered relations with anonymous attributes. All projections and dependencies will be specified with a positional calculus. The notion of type will be recovered from the inclusion dependencies in the schema.

A *relation schema* $R : k$ is a sequence of anonymous attributes with arity k. We will usually denote the arity of R as $|R|$. Given $1 \leq i \leq |R|$, R^i denotes the i-th attribute of R. A *tuple* with arity n is a sequence of values $\langle a_1, \ldots, a_n \rangle$. A *relation* r over a relation schema R is a set of tuples with arity $|R|$. Let $\mathbf{R} = \{R_1, \ldots, R_k\}$ be a set of relation schemas. A *database* over $\mathbf{R}$ is a set of relations $d = \{r_1, \ldots, r_k\}$, where each r_i is over R_i.

Let t be a tuple with arity n and $X = \langle i_1, \ldots, i_k \rangle$ a sequence of distinct natural numbers in $\{1, \ldots, n\}$, then $t[X]$ is the *tuple projection* $\langle a_{i_1}, \ldots, a_{i_k} \rangle$. The notion of projection can be extended to relations naturally: the *relational projection* $r[X]$ is just $\{t[X] \mid t \in r\}$. If $X = \langle 1, \ldots, |r| \rangle$, then $r[X]$ is the identity projection on r. In a relational projection we omit the braces around sequences: $r[i_1, \ldots, i_k]$ is identical to $r[\langle i_1, \ldots, i_k \rangle]$. In a database we will denote the relation over

a given relation schema R using the corresponding lower-case letter r.

A *functional dependency* (FD) over a set of relation schemas $\mathbf{R}$ is a statement of the form $R : X \rightarrow Y$, where $R \in \mathbf{R}$, and X and Y are sequences of distinct natural numbers in $\{1, \ldots, |R|\}$. A FD $R : X \rightarrow Y$ is satisfied in a database d, denoted by $d \models R : X \rightarrow Y$, whenever $\forall t_1, t_2 \in r$, if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$. A FD $R : X \rightarrow \langle 1, \ldots, |R| \rangle$ denotes a *superkey constraint*, and will be abbreviated as $R : X \rightarrow R$

An *inclusion dependency* (IND) over a set of relation schemas $\mathbf{R}$ is a statement of the form $R[X] \subseteq S[Y]$, where $R, S \in \mathbf{R}$, $|X| = |Y|$ and X, Y are sequences of distinct natural numbers in $\{1, \ldots, |R|\}$ and $\{1, \ldots, |S|\}$, respectively. When X is an identity projection we will denote $R[X]$ just by R. An IND $R[X] \subseteq S[Y]$ is satisfied in a database d, denoted by $d \models R[X] \subseteq S[Y]$, whenever $r[X] \subseteq s[Y]$.

Definition III.1. A *database schema* is a tuple $\langle \mathbf{R}, \Sigma_I, \Sigma_F \rangle$ where:

- $\mathbf{R}$ is a set of relation schemas.
- Σ_I is a set of inclusion dependencies over $\mathbf{R}$.
- Σ_F is a set of functional dependencies over $\mathbf{R}$.

A set of dependencies Σ is satisfied in a database d, denoted by $d \models \Sigma$, if $\forall \sigma \in \Sigma, d \models \sigma$. Σ logically implies a dependency σ, denoted by $\Sigma \models \sigma$, if whenever $d \models \Sigma$ then $d \models \sigma$. Let Σ^* denote the set of all FDs and INDs that are logically implied by Σ. Given a database schema $\langle \mathbf{R}, \Sigma_I, \Sigma_F \rangle$ it is well-known that it is possible to compute Σ_I^* using the axiom system from [3], and Σ_F^* using Armstrong's axioms [4]. However, if we consider the set of FDs and INDs together $\Sigma = \Sigma_I \cup \Sigma_F$, then computing Σ^* is undecidable [5], [6]. To retain decidability, we will use a rather strong notion of equality between database schemas:

Definition III.2. Two database schemas $\langle \mathbf{R}, \Sigma_I, \Sigma_F \rangle$ and $\langle \mathbf{R}', \Sigma_I', \Sigma_F' \rangle$ are equal if $\mathbf{R} = \mathbf{R}'$ and $\Sigma_I^* = \Sigma_I'^*$ and $\Sigma_F^* = \Sigma_F'^*$.

IV. A Static Alloy Subset

The objective of this section is to identify a static subset of Alloy that can be mapped into a database schema, and vice-versa. This subset will be denoted as Alloy^{DB}. As seen in the previous section, we only consider the most common integrity constraints on databases, namely, FDs and INDs. It is obvious that, by resorting only to these constraints, it is impossible to express all the power of Alloy's specification logic. Nonetheless, Alloy^{DB} is still expressive enough to cover many non-trivial examples, including almost all of our running example.

In fact, the only aspect that will not be covered in the example concerns signature hierarchy. In Alloy, different extensions of a signature are necessarily disjoint. Using just FDs and INDs it is not possible to state that two relations are disjoint, and thus it will not be possible to capture the precise semantics of **extends**. The same applies to abstract signatures, since we would need to express that some relation equals the reunion of others. We can however specify that one relation is contained in another, and thus our formal definition of Alloy^{DB} will include the keyword **in** that models a more relaxed notion of inheritance. In our example, this corresponds to the following alternative declaration of the objects of a filesystem:

sig *Obj* {
  name : **one** *Name*
}
sig *Dir, File* **in** *Obj* {}

Notice that there is no longer the guarantee that an object is either a directory or a file. Considering that the final target of our transformation is not just a database schema, but a fully materialized object-oriented implementation, this limitation is not relevant since the semantics of **extends** and **abstract** matches that of most object-oriented languages, and thus can easily be enforced in the application layer.

INDs allow us to specify inclusions between arbitrary projections of relations. In Alloy, projections can be specified by composing relations with signatures. For example, *parent·Dir* projects out the the last domain of the ternary $parent$ relation. To further project out the first domain we could perform another composition: *FS·(parent·Dir)*. Given that Alloy relations are ordered, and that projection is achieved via composition, in order to project out a middle domain it is necessary to first perform a permutation on the relation. For example, the binary relation between filesystems and parent directories could be defined as follows:

$\{$*f:FS, d:Dir, o:Obj* $\mid f \rightarrow o \rightarrow d$ **in** *parent*$\}$·*Obj*

Without loss of generality, a projection in Alloy can be specified by a tuple $\langle R, \alpha, \pi \rangle$, where R is any relation declared in a specification, α is a bijective function that specifies a permutation on R, and π is the number of projected-out domains on the right-hand side of R. π must be less then $|R|$, the arity of R. For example, the above projection can be succinctly specified as $\langle parent, \{1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 2\}, 1 \rangle$. Notice that it is rather trivial to parse this representation from more liberal syntactic representations of projections, namely those where we have projected domains on both the left- and right-hand side of a relation. When specifying permutations, we will sometimes denote any bijective function where i maps to j just by $i \mapsto j$, and the identity permutation by id. The identity projection $\langle R, id, 0 \rangle$ will be denoted just by R.

Informally, an Alloy^{DB} specification includes a set of signature and field declarations, an inclusion relation between signatures, and a set of facts of the form *R* **in** *S*, where R is any projection and S is any cartesian product of projections, optionally constrained by a **lone** multiplicity (to express a functional dependency). Formally, we have the following definition.

Definition IV.1. An *Alloy^{DB} specification* is a tuple $\langle \mathbf{S}, \trianglelefteq, \mathbf{F}, \Sigma \rangle$ where:

- $\mathbf{S}$ is a set of signature declarations.

- $\trianglelefteq$ is an acyclic and simple inclusion relation between signatures[1].
- **F** is a set of field declarations $F : S_1 \times \ldots \times S_{|F|}$, where F is the field name and each $S_i \in S$ is a signature.
- Σ is a set of facts $\langle\langle R, \alpha, \pi\rangle, \langle R_1, \alpha_1, \pi_1\rangle \times \ldots \times \langle R_k, \alpha_k, \pi_k\rangle, m\rangle$, each stating that a projection $\langle R, \alpha, \pi\rangle$, where $R \in S \cup F$, is contained in the cartesian product of any other k projections. Furthermore, $1 \leq m \leq k$ specifies a **lone** multiplicity constraint after the m-th projection. If $m = k$, this constrain is vacuous.

Furthermore, Σ must be consistent with $\trianglelefteq$, i.e, if $A \trianglelefteq B$ then the fact $\langle A, B, 1\rangle$ exists in Σ. Notice that π, the number of projected-out domains in the left-hand side of a fact, is redundant since type-checking guarantees that

$$\pi = |R| - \sum_{i=1}^{k}(|R_i| - \pi_i)$$

Although we require Σ to be consistent with $\trianglelefteq$, the latter cannot be specified just by facts: in Alloy, a top-level signature is disjoint from any other signature, and any fact specified in the model is type-checked against this premise. Namely, the following specification is not equivalent to the one above, and originates a type-error in Alloy, since neither *Dir* nor *File* can be contained in *Obj*.

sig *Obj* {
  name : **one** *Name*
}
sig *Dir, File* {}
fact {
  Dir **in** *Obj*
  File **in** *Obj*
}

In order to parse an Alloy specification into the above formal model, a field declaration must be split into a type declaration and one or more explicit facts, stating similar properties in our restricted syntactic form. For example, consider the field declaration *root :* **one** *(Dir & objects)*. From the conjunction it is easy to infer that $root : FS \times Dir$. Furthermore, the inclusion and multiplicity constraints could be expressed by the following facts:

root **in** *objects*
root **in** *FS* → **lone** *Dir*
root **in** *FS* → **some** *Dir*

The first two facts map directly into our syntactic form. In general, to capture **some** multiplicity constraints in a database schema we would need join dependencies [7], and thus they were excluded from the Alloy^{DB} specification. However, **some** constraints on binary relations can easily be specified using INDs. The last fact could be represented by the following equivalent one:

FS **in** *root·Dir*

[1]A relation $R : A \times B$ is simple if *R* **in** *A* → **lone** *B*

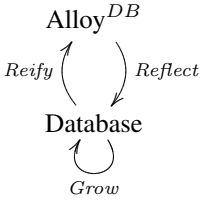
sig *Name* {}
sig *Obj* {
  name : **set** *Name*
}
sig *Dir, File* **in** *Obj* {}
sig *FS* {
  objects : **set** *Obj,*
  root : **set** *Dir,*
  parent : Obj → *Dir*
}
fact {
  Dir **in** *Obj*
  File **in** *Obj*
  name **in** *Obj* → **lone** *Name*
  Obj **in** *name·Name*
  {*n:Name,o:Obj* | *o*→ *n* **in** *name*} **in** *Name* → **lone** *Obj*
  root **in** *FS* → **lone** *Dir*
  FS **in** *root·Dir*
  root **in** *objects*
  parent **in** *objects* → **lone** *Dir*
  {*f:FS,d:Dir,o:Obj* | *f*→ *o*→ *d* **in** *parent*}·*Obj* **in** *objects*
}

Fig. 3. Alloy^{DB} specification of a filesystem.

Using similar straightforward techniques, our running example could be parsed into a formal model, whose transliteration into Alloy is presented in Figure 3.

V. Mapping between Alloy^{DB} and Databases

In this section, we formalize the mappings between Alloy^{DB} specifications and database schemas. The following mappings will be defined:



Reflect maps an Alloy^{DB} specification to a database schema. The backwards translation proceeds in two steps: first, the *Grow* transformation is applied in order to enforce that all distinct types in a database exist as unary tables; then the resulting schema is translated into Alloy^{DB} with the *Reify* translation. An equivalence notion on database schemas is also defined, and the transformations are proved correct against this definition.

A. Mapping Alloy^{DB} Specifications to Database Schemas

The definition of *Reflect* is quite simple. Signatures and fields have a one-to-one mapping with the relations in the database schema. Field types can be enforced in the database using unary INDs to the respective unary relations. Alloy^{DB} facts will originate one IND for each projection in the

cartesian product of the right-hand side. The specification of permutations using bijective functions makes the generation of projections trivial: we will use the notation $\langle 1, \ldots, i \rangle_\alpha$ as an abbreviation to $\langle \alpha(1), \ldots, \alpha(i) \rangle$; and α^{-1} to denote the inverse function of a bijective function α. **lone** multiplicities also possess a direct translation in terms of FDs. The formal definition of the mapping is as follows:

Definition V.1. Given an Alloy^{DB} specification, $\mathit{Reflect}(\langle \mathbf{S}, \trianglelefteq, \mathbf{F}, \Sigma \rangle) = \langle \mathbf{R}, \Sigma_I, \Sigma_F \rangle$ is a database schema where:

- $\mathbf{R}$ contains for each $S \in \mathbf{S}$ an ordered relation $S : 1$, and for each $F : S_1 \times \ldots \times S_{|F|} \in \mathbf{F}$ an ordered relation $F : |F|$.
- Σ_I contains:
  - For each field $F : S_1 \times \ldots \times S_{|F|} \in \mathbf{F}$, an unary IND for each $1 \leq i \leq |F|$:

$$F[i] \subseteq S_i$$

  - For each fact $\langle \langle R, \alpha, \pi \rangle, \langle R_1, \alpha_1, \pi_1 \rangle \times \ldots \times \langle R_k, \alpha_k, \pi_k \rangle, m \rangle \in \Sigma$, an IND for each $1 \leq i \leq k$:

$$R[1 + \textstyle\sum_{j=1}^{i-1}(|R_j| - \pi_j), \ldots, \sum_{j=1}^{i}(|R_j| - \pi_j)]_{\alpha^{-1}} \subseteq R_i[1, \ldots, |R_i| - \pi_i]_{\alpha_i^{-1}}$$

- Σ_F contains, for each fact $\langle \langle R, \alpha, \pi \rangle, \langle R_1, \alpha_1, \pi_1 \rangle \times \ldots \times \langle R_k, \alpha_k, \pi_k \rangle, m \rangle \in \Sigma$, a FD:

$$R : \langle 1, \ldots, \textstyle\sum_{j=1}^{m}(|R_j| - \pi_j) \rangle_{\alpha^{-1}} \rightarrow \langle 1 + \textstyle\sum_{j=1}^{m}(|R_j| - \pi_j), \ldots, |R| - \pi \rangle_{\alpha^{-1}}$$

  Notice that, when $m = k$, a trivial FD of the form $R : \langle 1, \ldots, |R| - \pi \rangle_{\alpha^{-1}} \rightarrow \langle\rangle$ is generated.

To illustrate the application of *Reflect*, Figure 4 presents the database schema that results from mapping the example from Figure 3. The trivial FDs are omitted.

From a database engineering perspective, the resulting schema is probably not the most intuitive. The main reason for this is that signatures, that realize the notion of type in Alloy, denote unary relations whose sets of atoms contain all existing instances of their corresponding types. Some of the unary relations created by *Reflect* when mapping signatures are redundant and can be safely deleted. That is the case of FS in our running example: since a filesystem must always have a root, the set of all existing filesystems is also contained in the $root$ relation (note that $root[1] \subseteq FS$ and $FS \subseteq root[1]$). From a database design perspective, it might make sense to delete all such unary relations. That is the approach followed, for example, in [8]. Nevertheless, this decision would affect the semantics of the resulting database in comparison to the original specification. We return to this discussion at the end of Section VI-A.

$$\begin{aligned}
\mathbf{R} = \{&Name : 1, Obj : 1, Dir : 1, File : 1, FS : 1,\\
&name : 2, objects : 2, root : 2, parent : 3\}\\
\Sigma_I = \{&Dir \subseteq Obj, File \subseteq Obj,\\
&name[1] \subseteq Obj, name[2] \subseteq Name,\\
&objects[1] \subseteq FS, objects[2] \subseteq Obj,\\
&root[1] \subseteq FS, root[2] \subseteq Dir\\
&parent[1] \subseteq FS, parent[2] \subseteq Obj, parent[3] \subseteq Dir,\\
&Obj \subseteq name[1], FS \subseteq root[1], root \subseteq objects,\\
&parent[1,2] \subseteq objects, parent[1,3] \subseteq objects\}\\
\Sigma_F = \{&name : \langle 1 \rangle \rightarrow \langle 2 \rangle, name : \langle 2 \rangle \rightarrow \langle 1 \rangle,\\
&root : \langle 1 \rangle \rightarrow \langle 2 \rangle, parent : \langle 1, 2 \rangle \rightarrow \langle 3 \rangle\}
\end{aligned}$$

Fig. 4. Database schema for the example of Figure 3.

B. Mapping Database Schemas to Alloy^{DB} Specifications

Not all database schemas can be mapped to Alloy^{DB}. In particular, it will be required that the schema is confluent.

Definition V.2. Given a database schema $\langle \mathbf{R}, \Sigma_I, \Sigma_F \rangle$, let $\subseteq_1$ be the unary fragment of the INDs in Σ_I^*. Two unary projections $R[i], S[j]$ have the *same type* iff there exists a projection $T[k]$ (the meet projection) such that $R[i] \subseteq_1 T[k]$ and $S[j] \subseteq_1 T[k]$.

Definition V.3. A database schema is *confluent* iff for all unary projections $R[i], S[j], T[k]$, if $R[i] \subseteq_1 S[j]$ and $R[i] \subseteq_1 T[k]$ then $S[j]$ and $T[k]$ have the same type.

As will be discussed in Section V-D, unless unions are allowed in the right-hand side of Alloy^{DB} facts, non-confluent schemas cannot be mapped to equivalent Alloy^{DB} specifications. Moreover, to simplify the presentation of this mapping, we will require the input schema to *Reify* to be *confluent to unary relations*.

Definition V.4. A database schema is *confluent to unary relations* iff it is confluent and the meet projection (in the same type definition) is restricted to be on an unary relation.

The following proposition is a trivial consequence of Alloy type checking mechanism, that disallows one to specify an inclusion between disjoint signatures, and the fact that an IND between each unary projection and an unary relation is created by *Reflect*.

Proposition V.1. *If s is an Alloy^{DB} specification then $\mathit{Reflect}(s)$ is confluent to unary relations.*

It is always possible to add redundant unary relations to a confluent schema in order to make it confluent to unary relations. That is the role of the *Grow* transformation.

Definition V.5. Given a confluent database schema, $\mathit{Grow}(\langle \mathbf{R}, \Sigma_I, \Sigma_F \rangle) = \langle \mathbf{R}', \Sigma_I', \Sigma_F' \rangle$ is a database schema where:

- $\mathbf{R}'$ contains all relations in $\mathbf{R}$ plus an unary relation $R^i : 1$ for each attribute R^i not contained in an unary relation.
- Σ_I' contains all INDs in Σ_I plus $R[i] \subseteq R^i$ and $R^i \subseteq R[i]$ for each R^i not contained in an unary relation.

- $\Sigma'_F = \Sigma_F$.

The added INDs guarantee that the new relations are indeed redundant. Again, it is trivial to prove that *Grow* achieves the desired effect.

Proposition V.2. *If s is a confluent database schema then $Grow(s)$ is confluent to unary relations.*

When a database schema is confluent to unary relations it is more simple to perform the mapping to an Alloy^{DB} specification, because all necessary signatures are guaranteed to exist as unary tables. Given an attribute R^i, let $\underline{R^i}$ be any unary relation where $R[i]$ is contained. On a schema confluent to unary relations $\underline{R^i}$ is guaranteed to exist for all R^i. *Reify* can now be defined as follows.

Definition V.6. Given a database schema confluent to unary relations, $Reify(\langle \mathbf{R}, \Sigma_I, \Sigma_F \rangle) = \langle \mathbf{S}, \trianglelefteq, \mathbf{F}, \Sigma \rangle$ is an Alloy^{DB} specification where:

- $\mathbf{S}$ contains all unary relations.
- $\trianglelefteq$ is any acyclic and simple fragment of $\subseteq_1$ that is complete in the sense that, if two unary tables have the same type under $\subseteq_1$ then the corresponding signatures have the same type under $\trianglelefteq$.
- $\mathbf{F}$ contains all relations $R : \underline{R^1} \times \ldots \times \underline{R^{|R|}}$, such that $R \in \mathbf{R}$ and $|R| > 1$.
- Σ contains:
  - For each IND $R[X] \subseteq S[Y]$ a fact

    $$\langle \langle R, \alpha, |R| - k \rangle, \langle S, \beta, |S| - k \rangle, 1 \rangle$$

    where $k = |X| = |Y|$ and α and β are any permutations such that $X_\alpha = Y_\beta = \langle 1, \ldots, k \rangle$.
  - For each FD $R : X \to Y$ a fact

    $$\langle \langle R, \alpha, |R| - j \rangle, \langle \underline{R^{\alpha(1)}}, id, 0 \rangle \times \ldots \langle \underline{R^{\alpha(j)}}, id, 0 \rangle, i \rangle$$

    where $i = |X|$, $j = |X \cup Y|$, and α is any permutation such that $X_\alpha = \langle 1, \ldots, i \rangle$ and $(Y - X)_\alpha = \langle i + 1, \ldots, j \rangle$.

Once again we have a one-to-one mapping between relations in the schema and signatures and fields. We do not commit to a particular strategy when inferring $\trianglelefteq$, and just give a sufficient condition for it to be correct. A possible implementation that guarantees this condition could be:

1) Build a graph where nodes are unary relations and edges are INDs between them.
2) Compute the strongly connected components (SCCs) of this graph.
3) For each terminal SCC pick one relation to serve as top-level signature.
4) For all the remaining relations add a pair to $\trianglelefteq$ stating in which top-level signature it is contained.

Thanks to permutations, INDs and FDs have a direct translation to Alloy^{DB}. However, many redundant facts are generated, namely those corresponding to the INDs $R[i] \subseteq \underline{R_i}$, that assert the connection between the attributes of a relation and the signatures that contain them (they are redundant because the field declaration enforces them). In a concrete implementation of *Reify* it is trivial to remove them from the final specification. In the case of a FD $R : X \to Y$, where Y is not disjoint from X, we first remove the trivial dependencies from Y: the fact inserted in the specification corresponds to $R : X \to (Y - X)$.

C. Correctness of the Mappings

To be able to check that our mappings are correct, we first need to define a notion of equivalence between database schemas that is more relaxed than the equality presented in definition III.2. Consider a database schema with a single relation schema $R : 2$. After applying *Reify*, we obtain an Alloy^{DB} specification equivalent to the following, where A and B correspond to the unary relations introduced by *Grow* to make the schema confluent to unary relations:

```
sig A { r : set B}
sig B {}
fact {
  A in r·B
  B in A·r
}
```

If we now apply *Reflect*, we end up getting a different schema with three relations $R : 2, A : 1, B : 1$ plus the INDs $A \subseteq R[1], R[1] \subseteq A, B \subseteq R[2], R[2] \subseteq B$. Clearly, the single attributes of both A and B are redundant, in the following sense [9]:

Definition V.7. Given a database schema $\langle \mathbf{R}, \Sigma_i, \Sigma_F \rangle$, the i-th attribute of a relation schema $R \in \mathbf{R}$ is *redundant* if, whenever d is a database over $\mathbf{R}$ which satisfies Σ, then, for every tuple $t \in r$, if $t[i]$ is replaced by a value $v \neq t[i]$ then the resulting database does not satisfy Σ.

Ideally, we would like our notion of equivalence to ignore all redundant attributes, by converting first the schemas to the so called *Attribute Redundancy Free Normal Form* [9], but it is currently unknown how to do that in general. However, in this particular case, it suffices to eliminate redundant unary relations, using the following procedure:

Definition V.8. Given a database schema $\Gamma = \langle \mathbf{R}, \Sigma_I, \Sigma_F \rangle$, $Reduce(\Gamma)$ is the schema that results from exhaustively applying the following reduction rule: if R is an unary relation such that $\exists S \in \mathbf{R}, 1 \leq i \leq |S|, S \neq R \wedge R \subseteq_1 S[i] \wedge S[i] \subseteq_1 R$, then remove R from $\mathbf{R}$ and redirect all INDs pointing to R to $S[i]$.

Definition V.9. Two database schemas $\Gamma = \langle \mathbf{R}, \Sigma_I, \Sigma_F \rangle$ and $\Delta = \langle \mathbf{R}', \Sigma'_I, \Sigma'_F \rangle$ are equivalent, written $\Gamma \equiv \Delta$, iff $Reduce(\Gamma) = Reduce(\Delta)$.

It is trivial to show that *Grow* generates an equivalent schema:

Proposition V.3. *Given any database schema $\Gamma = \langle \mathbf{R}, \Sigma_I, \Sigma_F \rangle$, $Grow(\Gamma) \equiv \Gamma$.*

The following lemma is central to establish the correctness of the mappings:

Lemma V.4. *Given any database schema* $\Gamma = \langle \mathbf{R}, \Sigma_I, \Sigma_F \rangle$ *confluent to unary relations,* $Reflect(Reify(\Gamma)) \equiv \Gamma$.

Proof: First notice that, in both mappings, unary relations are in one-to-one correspondence to signatures and non-unary ones in one-to-one correspondence to fields. Thus, the resulting set of relation schemas is equal to $\mathbf{R}$. Each IND gives origin to a single Alloy^{DB} fact, that is translated back to exactly the same IND. Field declarations will originate a set of unary INDs of the form $R[i] \subseteq \underline{R^i}$, but by definition of $\underline{R^i}$ they follow from Σ_I and thus are redundant. A FD $R : X \rightarrow Y$, where X and Y are disjoint, is translated to a single Alloy^{DB} fact, that will be translated back into the same FD plus a set of redundant unary INDs of the form $R[i] \subseteq \underline{R^i}$. On the other hand, a FD $R : X \rightarrow XY$ will be translated back as $R : X \rightarrow Y$, which is equivalent to the original by the Armstrong's axioms [4]. ∎

The correctness of the mapping from database schemas to Alloy^{DB} is a corollary of this lemma and the previous proposition:

Theorem V.5. *Given any confluent database schema* $\Gamma = \langle \mathbf{R}, \Sigma_I, \Sigma_F \rangle$, $Reflect(Reify(Grow(\Gamma))) \equiv \Gamma$.

To show the correctness of the other mapping, we also need a notion of equivalence between Alloy^{DB} specifications. Here, we will simply resort to the database schema equivalence via the *Reflect* mapping:

Definition V.10. Two Alloy^{DB} specifications $\Gamma = \langle \mathbf{S}, \trianglelefteq, \mathbf{F}, \Sigma \rangle$ and $\Delta = \langle \mathbf{S}', \trianglelefteq', \mathbf{F}', \Sigma' \rangle$ are equivalent, written $\Gamma \equiv \Delta$, iff $Reflect(\Gamma) \equiv Reflect(\Delta)$.

This notion of equivalence makes the correctness of the mapping from Alloy^{DB} to databases a trivial corollary of the previous results:

Theorem V.6. *Given any Alloy^{DB} specification* $\Gamma = \langle \mathbf{S}, \trianglelefteq, \mathbf{F}, \Sigma \rangle$, $Reify(Reflect(\Gamma)) \equiv \Gamma$.

Notice that, due to proposition V.1, it is not necessary to apply the *Grow* transformation on the resulting database schema.

D. Non-confluent Database Schemas

As we have seen, the database schemas equivalent to Alloy^{DB} specifications are confluent to unary relations. Using the *Grow* transformation we can handle any confluent database schema, but would it be possible to map non-confluent schemas to Alloy? Consider, for example a database schema with three unary relations, A, B and C, and the INDs $A \subseteq B, A \subseteq C$. This is the simplest schema that breaks the confluence property: since A is contained in B and C, then these attributes should have the same type. One possible way to map this schema to Alloy would be to force the confluence by adding an artificial superset signature including both B and C:

```
sig T {}
sig A,B,C in T {}
fact {
  A in B
  A in C
  T in B + C
}
```

Since A cannot inherit from both B and C, it is declared as a subset signature of T and later constrained to be in both B and C. Additionally, we must restrict the domain of T to be in the union of B with C, otherwise the Alloy specification would allow values not present in the original database schema. We name this constraint an union constraint.

Without explicit support for union constraints on the database side, it is not trivial to define a general technique to map Alloy specifications into databases: the naive approach of not mapping signatures that are contained in a union constraint will not work in general. Consider the following Alloy specification:

```
sig T {}
sig A,B in T {}
fact {
  T in A + B
}
```

In this case, if T is not mapped to the database schema, we will be left with two unrelated unary relations. Since there is no proof that A and B have the same type, converting the resulting database schema back into Alloy would originate a specification completely different from the original:

```
sig A,B {}
```

If union constraints were supported on database schemas (as generalization of INDs), it would be quite simple to extend our transformations to work correctly. In fact, since T is redundant according to definition V.7, we would just need to extend *Reduce* to remove also this kind of redundancy, in order to get a suitable equivalence definition on database schemas. The problem is that this kind of constraint is non-standard, and further research is needed to determine how can they be enforced on databases and how do they interact with the other dependencies.

VI. Object-Oriented Application Layer

As discussed in [10], Alloy is an object modelling notation, albeit more in the sense of a data modelling language for describing the conceptual entities of software systems and their relationships. At the same time, it has an object-oriented flavor, incorporating features such as abstraction and inheritance, and the most standard Alloy idioms also have an inherent notion of state over which the dynamic aspects of a system are specified.

Building on this duality, in this section we debate how an Alloy specification can be mapped into an object-oriented application layer, with the overall system state fully materialized in an underlying relational database (as prescribed in Figure 1). This mapping encompasses two steps: the translation of a static

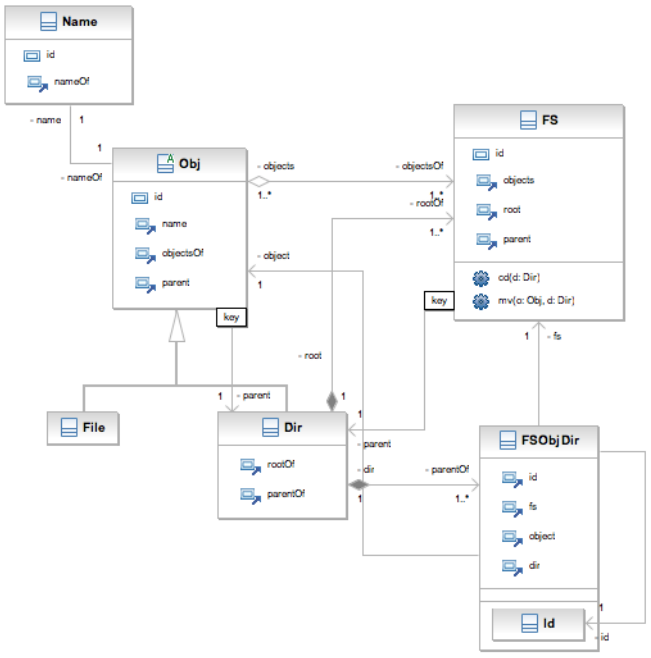


Fig. 5. UML class diagram for the filesystem from Figure 2.

Alloy^{DB} data model, enriched with abstraction and extension, to a Java class hierarchy that is made persistent through an off-the-shelf ORM framework [11]; and the conversion of the dynamic operations in the specification to methods in the corresponding classes. The UML class diagram from Figure 5 illustrates the resulting object-oriented schema for the filesystem example (Figure 2).

A. Mapping the Static Data Model

The signature hierarchy in an Alloy specification has a direct correspondence with the class hierarchy in the object-oriented implementation. For each signature S, we create a class S with an identifier of the primitive type **int** (to be used by the ORM). The class S is made abstract or an extension of another class S' (inheriting its identifier) if the signature S is abstract or extends a signature S', respectively. As an exception, if S extends Alloy's built-in signature **Int**, we do not create a class S and all the instances of S have the primitive type **int**. It is also possible to map other signatures to primitive types: for example, the *Name* signature in the filesystem example of Figure 2 is a natural candidate to be mapped into a string.

The fields in an Alloy specification, relating signatures in the model, originate associations between classes, to be expressed as specific attributes of each class. Instead of Alloy's undirected relationships, where for some relation r **in** $A \rightarrow B$ we can use both $a{\cdot}r$ and $r{\cdot}b$ to select related elements, here we must conform to a directed navigational style, and create an attribute r in both A and B to navigate between them. Of course, the ORM must guarantee the bidirectionality of the association. When a signature is mapped to a primitive type, its associations will no longer be bidirectional: if we see names as strings in our example, the relation *name* between filesystem objects and names will only be represented in the first class.

Thus, for each field declaration $F : S_1 \times \ldots \times S_{|F|}$, we insert attributes in the class of each signature S_i in F, unless S_i is mapped to a primitive type. The name and type of the attributes depends on the size of the field and on the **lone** multiplicities involving S_i (represented as FDs in an underlying database d):

- if $|F| = 2$ and $d \models F : \langle S_i \rangle \rightarrow \langle S_j \rangle$, where $i \neq j$, insert an attribute named F with type S_j; otherwise, insert a collection of objects S_j named F;
- if $|F| > 2$ and $d \models F : \langle S_i \rangle \rightarrow F$ insert an attribute of type S_j for each signature S_j in $F - \{S_i\}$;
- if $|F| = 3$ and $d \models F : \langle S_i, S_j \rangle \rightarrow \langle S_k \rangle$, where $i \neq j \neq k$, insert a map from S_j to S_k named F;
- otherwise, insert a collection named F whose instances belong to an association class that represents the entire field.

As an optimization, we remove redundant unary relations from the database schemas, according to the *Reduce* transformation from Section V-C. The ORM must take these redundancies into account in order to make the corresponding classes persistent. Also, to regulate the creation of non-redundant signature relations, we decided to delete unary relations for signatures extending built-in signatures, such as **Int**, or any other signature mapped to a primitive type.

Note that, at this point, further optimizations are likely to be performed in a normal database development cycle. Recalling the database schema from Figure 4, a database designer could, for instance, choose to fuse the $objects$ and $parent$ relations into a new database table with a nullable parent field. By resorting to a *table-per-class* strategy (we implicitly use *table-per-subclass*), the designer could also alter the objects hierarchy into a single table with a boolean type discriminator column, testifying if an object is either a directory or a file. However, these performance-improving techniques normally favor schema denormalization and can compromise the consistency of the database. They are outside the scope of an automated mapping tool, and should later be assessed and applied by developers in the generated implementations.

B. Mapping the Dynamic Operations

The dynamic operations are encoded as methods in the object-oriented schema and have a one-to-one correspondence with the stateful predicates in the original Alloy specification. A predicate is called stateful if it contains at least one pair of variables that have the same type and share the same name with and without priming (e.g., f and f'). The type of each variable is a signature, instead of an arbitrary unary relation as in full Alloy.

For each stateful predicate, we declare a method with the same name in the corresponding class. The method declaration contains one argument for each non stateful variable and one argument for each other pair of states. Note that so far we only generate the declarations, but do not synthesize the body of methods.

VII. Implementation

We have implemented a prototype tool for translating between Alloy specifications and database schemas, according to the mappings presented in Section V. In the forward direction, the tool uses an Alloy parser as a front-end and outputs a SQL database together with a Java application layer. With this prototype, we have successfully generated for our running example a database schema and the corresponding object-oriented application layer. We have also manually implemented and tested different properties of the ORM gluing code. The algorithm for the automatic generation of the ORM code is ongoing work.

The implementation differs from the proposed mappings in some technical details:

a) Naming: The SQL language considers unordered relations with attribute names, in contrast to the database model from Section III. Thus, the implementation assumes a default order for relations and creates new names for attributes based on their signature domain. Likewise, SQL is case-insensitive whether Alloy is case sensitive, from what some renaming of relations may also be necessary.

b) Key constraints: In SQL, only superkey constraints can be expressed via primary key or unique constraints. All non-superkey FDs are consequently ignored.

c) Referential constraints: The SQL92 standard defines foreign keys as many-to-one relationships, meaning that a relation must only refer to superkeys, while INDs can refer to arbitrary projections. Consequently, this imposes a non-syntactic restriction disallowing references to non-keys: our tool recognizes and rejects these cases and alternatively allows the generation of non-standard SQL triggers. Above that, the INDs in the database schema are possibly circular, entailing circular foreign keys. The traditional solution to this problem is to declare circular foreign keys as deferrable, as prescribed in the SQL92 standard but unfortunately not supported by all RDBMS implementations. A workaround would be to remove the circularities in the database and enforce them as one-to-one associations in the ORM.

VIII. Related Work

The comparison between Alloy and UML, being UML the *de facto* modeling language in industry, is a natural topic ever since the genesis of Alloy [10], [12]. While Alloy is targeted at high-level abstract modeling, UML is a language for modeling complete object-oriented software systems at different levels of abstraction. As a result, while Alloy is concise and precise, UML tends to be complicated and ambiguous. The connection between both languages was explored in UML2Alloy [13], a tool that transforms UML class diagrams with OCL constraints to Alloy. Several methodologies (e.g. [14]–[16]) have been proposed for relational database modeling with UML. In general, they extend UML with non-standard stereotypes to model to the various ER diagram components, thus providing a similar expressive power and suffering from the same criticisms, such as difficulty to express the business rules and ease of inserting non-syntactic errors. The Alloy^{DB} fragment also has a similar expressive power, but allows the user to express the various dependencies amongst data in a more natural and uniform way, using a language amenable to fully automated verification.

Focusing on the data-modeling aspects, Alloy essentially realizes the notion of a *semantic data model* [17], by allowing us to model databases in terms of an (almost) natural language with entities and relationships between them. In [8], the authors study the formal connection between semantic database models and the relational model, and define mappings between the two using the Iris data model as a concrete case study. Apart from the source specification language, the main difference to our work is the type of constraints supported: while we restrict ourselves to the conventional inclusion and functional dependencies, they only allow unary inclusions but support other powerful constraints, such as join dependencies. It would be interesting to incorporate these in Alloy^{DB} since they can capture the **some** multiplicity.

Focusing now on tools/frameworks to derive implementations from Alloy specifications, the most relevant is without any doubt Alchemy [18], a tool that translates an Alloy specification into a PLT Scheme implementation that executes against a persistent database. The database integrity constraints are not deployed explicitly in the schema, but instead dynamically enforced by the generated implementation: the algorithm chooses non-deterministically which repair actions to execute, backtracking whenever a choice leads to an inconsistent database. Although more flexible, this run-time approach has several disadvantages: performance is one of them, but the most relevant is that the generated code is essentially a generic integrity repair black-box, with no meaningful connection to the original specification, and thus of little use to serve as the basis for a final implementation. Although we cover a much smaller subset of Alloy, we propose a compile-time approach that generates implementations that explicitly reflect the facts in the original specification.

In [19], a stateful Alloy idiom similar to the one used in Alchemy is used as starting point for the generation of Java executable code for embedded systems. Besides their structural Java translation, that resembles the one generated in our application layer, they encode Alloy built-in expressions using a dedicated Java library that mimics the behavior of Alloy's primitives. Unfortunately, the generated code does not reflect a natural object-oriented implementation, and thus this approach will not be followed in our future implementation. In the next section we discuss some alternatives.

Our work also has some similarities with WebAlloy [20], a framework for generation of policy-rich websites, where the data model and access policies are specified using Alloy. A WebAlloy specification can be decoupled into three main components: a static database model, a security layer with the access policies, and an user interface. Likewise to our approach of embodying (as much as possible) the business rules as integrity constraints on the database, the access policies denote pre- and post-conditions on tuple operations and can be seen as the check conditions for generalized integrity

constraints on the data model. This promotes the independence of the database and prevents applications from violating the rules/policies and, therefore, the consistency of the database. Our generated object-oriented application layer corresponds to one possible user interface.

IX. Concluding Remarks

In this paper, we have laid out the foundations for a full fledged framework to transform Alloy specifications into database-intensive applications. We have identified a static subset of Alloy that is equivalent to confluent database schemas enriched with functional and inclusion dependencies, and developed mappings from one formalism to the other. We have also shown how the confluence restriction can be lifted when mapping database schemas into Alloy, and how an object-oriented application layer can also be derived from the original specification. Objects in this layer are made persistent using a standard ORM framework and will be the target of the dynamic aspects of the specification.

A lot of work remains to be done to make the framework fully operational. First, we need to characterize more precisely the syntactic fragment of Alloy that corresponds to Alloy^{DB}: as our example shows, this fragment is definitively larger than the direct transliteration of Alloy^{DB} facts, but its exact expressiveness is still not clear. Second, we need to automate the generation of the ORM code. Third, we must settle on a strategy to translate the pre- and post-conditions that specify the operations, since so far only the method headers are being generated in the application layer. We envisage two possible (complementary) strategies for this: 1) adopt a design-by-contract methodology, and annotate the Java methods with the corresponding JML pre- and post-conditions [21]; 2) infer actual implementations from the post-conditions. Likewise to Alchemy, a procedure on the database side would be needed to restore integrity, since most of the time post-conditions are just partial specifications. For this to be viable, the resulting database schema should be free of insertion and update anomalies, which is quite difficult to guarantee when mixing INDs and FDs due to non-trivial interactions between both sets of dependencies. One possibility we are researching at the moment, is to restrict the database schemas to be in the so-called *Inclusion Dependency Normal Form* [9], a redundancy-free normal form where automatic integrity repair is feasible. The impact that this restriction has on the expressiveness of the source Alloy must be accessed carefully.

We are particularly interested in applying the developed mappings in the reengineering of legacy databases. By resorting to existing transformation frameworks [22], [23], we are researching the possibility of making them fully bidirectional [24], so that any changes made on the specification level could be merged back into the database design. Ideally, we would also like to derive the corresponding data migration functions.

References

[1] D. Jackson, *Software Abstractions: logic, language, and analysis*. The MIT Press, 2006.

[2] D. Jackson and J. Wing, "Lightweight formal methods," *IEEE Computer*, vol. 29, no. 4, pp. 21–22, 1996.

[3] M. Casanova, R. Fagin, and C. Papadimitriou, "Inclusion dependencies and their interaction with functional dependencies," in *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. ACM Press, 1982, pp. 171–176.

[4] W. W. Armstrong, "Dependency structures of database relationships," in *Proceedings of the IFIP Congress*, 1974, pp. 580–583.

[5] J. Mitchell, "The implication problem for functional and inclusion dependencies," *Information and Control*, vol. 56, no. 3, pp. 154–173, 1983.

[6] A. Chandra and M. Vardi, "The implication problem for functional and inclusion dependencies is undecidable," *SIAM Journal on Computing*, vol. 14, no. 3, pp. 671–677, 1985.

[7] J. Rissanen, "Theory of relations for databases – A tutorial survey," in *Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science*, ser. LNCS, vol. 64. Springer, 1978, pp. 536–551.

[8] P. Lyngbaek and V. Vianu, "Mapping a semantic database model to the relational model," *ACM SIGMOD Record*, vol. 16, no. 3, pp. 132–142, 1987.

[9] M. Levene and M. Vincent, "Justification for inclusion dependency normal form," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 281–291, 2000.

[10] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 2, pp. 256–290, 2002.

[11] C. Bauer and G. King, *Java Persistence with Hibernate*. Manning Publications, 2006.

[12] D. Jackson, "A comparison of object modelling notations: Alloy, UML and Z," August 1999, unpublished manuscript.

[13] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A challenging model transformation," in *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, ser. LNCS, vol. 4735. Springer, 2007, pp. 436–450.

[14] E. Nailburg and R. Maksimchuk, *UML for database design*. Addison-Wesley, 2001.

[15] M. Alalfi, J. Cordy, and T. Dean, "SQL2XMI: Reverse Engineering of UML-ER Diagrams from Relational Database Schemas," in *Proceedings of the 15th Working Conference on Reverse Engineering*. IEEE Computer Society, 2008, pp. 187–191.

[16] E. Marcos, B. Vela, and J. Cavero, "A methodological approach for object-relational database design using UML," *Software and Systems Modeling*, vol. 2, no. 1, pp. 59–72, 2003.

[17] M. Hammer and D. McLeod, "The semantic data model: a modelling mechanism for data base applications," in *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data*. ACM Press, 1978, pp. 26–36.

[18] S. Krishnamurthi, K. Fisler, D. Dougherty, and D. Yoo, "Alchemy: transmuting base Alloy specifications into implementations," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press, 2008, pp. 158–169.

[19] R. R. Ferreira, "Automatic code generation and solution estimate for object-oriented embedded software," in *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. ACM Press, 2008, pp. 909–910.

[20] F. Chang, "Generation of Policy-Rich Websites From Declarative Models," Ph.D. dissertation, Massachusetts Institute of Technology, 2009.

[21] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212–232, 2005.

[22] A. Bohannon, B. Pierce, and J. Vaughan, "Relational lenses: a language for updatable views," in *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM Press, 2006, pp. 338–347.

[23] P. Berdaguer, A. Cunha, H. Pacheco, and J. Visser, "Coupled schema transformation and data conversion for XML and SQL," in *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages*, ser. LNCS, vol. 4354. Springer, 2006, pp. 290–304.

[24] P. Stevens, "A landscape of bidirectional model transformations," in *Generative and Transformational Techniques in Software Engineering II*, ser. LNCS, vol. 5235. Springer, 2008, pp. 408–424.