Automated Synthesis and Dynamic Analysis of Tradeoff Spaces for Object-Relational Mapping

Hamid Bagheri, Chong Tang, and Kevin Sullivan

Abstract—Producing software systems that achieve acceptable tradeoffs among multiple non-functional properties remains a significant engineering problem. We propose an approach to solving this problem that combines synthesis of spaces of design alternatives from logical specifications and dynamic analysis of each point in the resulting spaces. We hypothesize that this approach has potential to help engineers understand important tradeoffs among dynamically measurable properties of system components at meaningful scales within reach of existing synthesis tools. To test this hypothesis, we developed tools to enable, and we conducted, a set of experiments in the domain of relational databases for object-oriented data models. For each of several data models, we used our approach to empirically test the accuracy of a published suite of metrics to predict tradeoffs based on the static schema structure alone. The results show that exhaustive synthesis and analysis provides a superior view of the tradeoff spaces for such designs. This work creates a path forward toward systems that achieve significantly better tradeoffs among important system properties.

Index Terms—Specification-driven Synthesis, Tradespace Analysis, ORM, Static Analysis, Dynamic Analysis, Relational logic.

1 Introduction

When the consequences of variations in design decisions leading to system implementations are unclear, it is important to conduct tradeoff studies. Design space exploration and tradeoff analysis can help decision-makers by revealing designs that people might miss [4], [24], [37], illuminating sensible and non-sensical tradeoffs, and helping decision-makers to balance tradeoffs that design decisions impose on diverse stakeholders. Ultimately it can provide evidence in support of principled decisions about which path or paths to pursue toward a realized system. Such studies can be done at different modeling and measurement granularities, for whole systems or individual components, and at many points in system development and evolution, and even runtime.

Yet, today, systematic tradeoff analysis remains rare. Instead, developers are usually trusted to use design heuristics, tacit knowledge, and other such methods in developing point solutions that, it is hoped, will be good enough for stakeholders in all key dimensions.

Similarly, when tools automatically produce implementations, they often use single-point strategies. Consider object-relational mapping (ORM) tools, now provided in many programming environments. They map object-oriented data models to relational schemas and code for managing application data. They often use a single mapping strategy, and do not help engineers to understand available solutions or the tradeoffs in time and space performance, evolvability, etc. that they entail.

In practice, systematic tradespace analysis is hard. Generating large numbers of complex variants by hand is often impractical. Spaces of variants can be huge, even for modest specifications. Property estimation functions can be hard to specify, validate, implement and compute, and can vary greatly in accuracy.

In this paper, we present an approach to solving this problem that combines synthesis of spaces of design alternatives and dynamic analysis of each point in the resulting spaces. The approach relies on specification-driven synthesis of both design spaces and test loads for comparative, dynamic analysis of non-functional properties of variant designs across such spaces.

We use formal notations, namely domain-specific modeling languages embedded in a relational logic, to specify design spaces. We then equip each language with a formal semantics in the form of a set of "non-deterministic mapping relations," which we also express in the underlying relational logic. The key idea is that these rules map a given model to a broad space of possible solutions, which vary in dimensions built into the semantics, and which are rooted in the given domain of discourse. We then subject synthesized designs (in the semantic domain) to dynamic measurement in multiple dimensions of performance under the synthesized loads. The result is a multi-dimensional, empirical characterization of the tradespace for the given model.

We hypothesize that this approach has potential to help engineers understand important tradeoffs among dynamically measurable properties for important classes of software designs, at meaningful scales, within reach of existing synthesis tools.

To test this hypothesis, we developed tools to enable a set of experiments in the domain of relational databases for object-oriented data models. For each of several data models, we used our approach to empirically characterize tradeoffs among designs predicted by previously published metrics based on model (namely SQL schema) structure as opposed to dynamic profiling. The designs we assessed were predicted to be high performing in the sense of being Pareto optimal in the dimensions measured. We then used our dynamic analysis results

to statistically assess the predictive accuracy of the previously published "static" metrics.

These static analysis functions estimate various properties from the static structures of relational schemas. Such functions can be computed nearly instantaneously, but their accuracy had not previously been assessed. Their suitability as a basis for making tradeoff decisions was thus unclear.

To summarize, the main contribution of this paper is an experimental demonstration that our approach to specification-driven synthesis and tradeoff analysis has the potential to better inform tradeoff decisions for important components of modern software systems. At a finer grain, this paper makes several component contributions:

- *Formal model of object-relational mapping strategies:* We develop what is to our knowledge the first formalization of fine-grained and mixed ORM strategies by means of mapping functions. We construct this formal specification in Alloy's relational logic [30].
- *Tradespace synthesis:* We contribute a fully automated approach showing how to exploit the power of our formal specification for derivation of formally precise and application-specific ORM tradeoff spaces.
- *Dynamic analysis of synthesized tradespaces:* We measure properties of synthesized database designs by synthesizing, from the same object models, schema-specific test loads and by then profiling database performance under these synthesized loads. We have not yet explored systematically the impact of variation in load characteristics as an independent variable, but our approach would in principle allow for such analysis.
- *Trademaker tool implementation:* We develop Trademaker as a web-accessible tool, backed with our formal analysis engine, for specification-driven synthesis of ORM tradespaces. We make Trademaker available to the research and education community[1].
- *Experimental validation of published ORM metrics:* We present results from experiments run on several case studies adopted from different domains, corroborating inaccuracy of published static metrics for predicting tradeoffs based on the schema structures alone. Data collected from the experiments further show that exhaustive synthesis and analysis provides a superior view of the tradeoff spaces for such designs.

The rest of this paper is organized as follows. Section 2 presents object-relational mapping as a concrete driving problem. Section 3 presents the overview of our approach. Sections 4–7 describe the details of our approach and its implementation for expression and synthesis of design spaces and loads. Section 8 and 9 report and discuss data from the experimental testing of the approach. Finally, the paper concludes with an outline of the related research and the future work.

1. http://www.jazz.cs.virginia.edu:8080/Trademaker

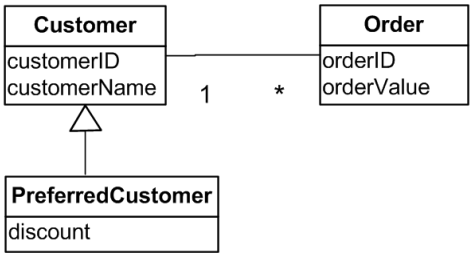

Fig. 1: A simple object model with three classes, $Order$, $Customer$, and $PreferredCustomer$, a one-to-many association between $Customer$ and $Order$, and with *PreferredCustomer* inheriting from $Customer$.

2 Driving Problem

While our long-term aim is to improve engineering productivity and quality through advances in design space science and technology, our short-term research strategy is to use the analysis of spaces of object-relational database mappings, in particular, as a tractable and useful *driving problem*.

It is common, nowadays, for software applications written in an object-oriented language, such as Java, to use relational databases for persistent storage. Transformations between instance models in these two paradigms yet encounter the so-called *object-relation impedance mismatch* problem [29], due to a significant distance between the object-oriented and relational theories. Dedicated object-relational middleware is now widely used to bridge the gap between object-oriented applications and relational database management systems (RDBMS). The bridge is realized on the basis of a data mapping specification.

Even if this approach relieves the developer of responsibility for the majority of runtime related aspects, e.g., storing and retrieving persistent objects, the development of application-specific data mapping specifications yet remains an inherently difficult and error-prone task.

In practice, one starts with an object-oriented data model (OODM) as in Figure 1 and eventually selects one of many possible strategies for mapping such a model to a relational database schema. Figure 2 illustrates three such strategies; and Listing 1, a database set-up script derived from one of these solutions. The developer is faced with the challenge of selecting from a wide variety of mapping strategies available for each class association and inheritance relationship. These mapping strategies have varying impacts on the non-functional properties of applications, such as time and space performance and evolvability [28], [31], [35].

Simple ORM solutions, many in everyday use, lock one into point solutions by using a single mapping strategy, failing to consider many possible available solutions or the tradeoffs that they entail [42]. The problem for the developer, then, becomes selecting the mapping strategies that best suit the desired non-functional requirement priorities for a particular application. It requires, among

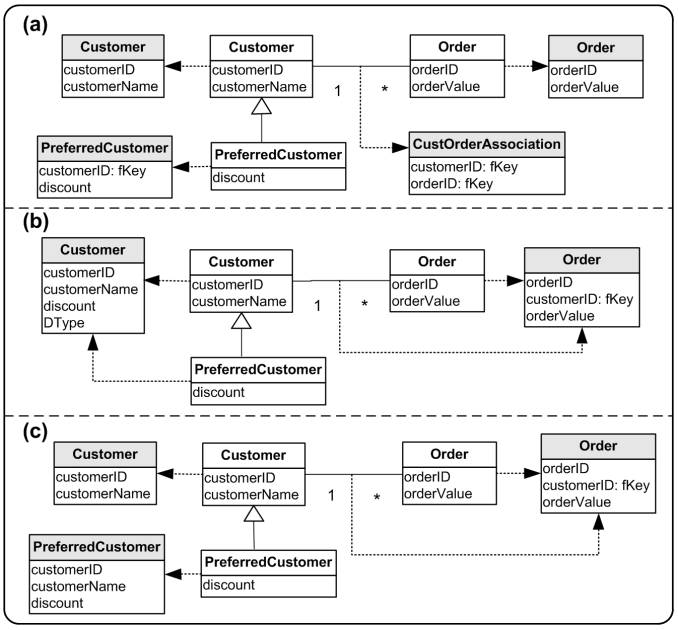

Fig. 2: Three mapping strategies. White boxes represent classes; gray titles, corresponding tables, and black and white arrows, mapping and inheritance relationships. Foreign keys are marked as *fKeys*.

```
1  CREATE TABLE `Order` (
2  `orderID` int(11) NOT NULL,
3  `orderValue` int(11),
4  `customerID` int(11),
5  KEY `FK_customerID_idx` (`customerID`),
6  PRIMARY KEY (`orderID`)
7  );
8
9  CREATE TABLE `Customer` (
10 `DType` varchar(31),
11 `discount` int(11),
12 `customerID` int(11) NOT NULL,
13 `customerName` varchar(31),
14 PRIMARY KEY (`customerID`)
15 );
```

Listing 1: Synthesized MySQL database creation script derived from the mapping alternative shown in Fig. 2b (elided for space and readability).

other things, a thorough understanding of both object and relational paradigms, of large spaces of possible mappings, and of the tradeoffs involved in making choices in these spaces.

3 Approach Overview

This section provides an overview of our technical approach for tradespace synthesis and dynamic analysis.

Figure 3 gives a high-level overview of Trademaker. Boxes represent processing modules, and ovals represent module inputs and outputs. Trademaker takes as input an application object model. Object models are given as expressions in *AlloyOM*, a domain-specific language embedded in Alloy's relational logic language [30]. To an input object model, it applies a *design space synthesis* function that, in essence, implements a non-deterministic

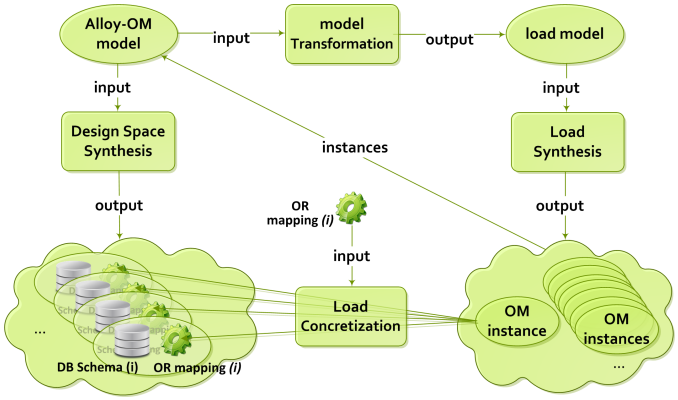

Fig. 3: Overview of Trademaker.

semantics to compute the set (or space) of concrete design variants from which we want to choose a design to achieve desirable tradeoffs. The design space synthesis function is realized as a semantic mapping predicate in relational logic, taking expressions in the object data modeling language to corresponding concrete design spaces in the semantic domain, here relational databases schemas. A relational logic solver computes the results, which are then transformed into useful forms as SQL database creation scripts (cf. Listing 1).

To dynamically analyze diverse database designs, we face the challenge that variant designs (such as different SQL schemas for the same object model) present different interfaces to the surrounding ORM tool. There are of course many commercial tools for generating database testing loads from schemas. In our case, however, such variant schemas implement a common object model. To provide a fair comparison of designs notwithstanding their particular exposed interfaces, a common test load needs to be specialized for each particular design. Fully automating the specialization of a common test load for fair comparative profiling of varying design solutions posed an interesting technical challenge.

Trademaker thus first automatically transforms the object model into a corresponding load model that drives synthesis of common test loads, which are essentially object model data instances (*OM-instances* in Fig. 3). We call them common loads because all concrete database implementations would have to handle them. They are object-model-level, rather than concrete-database-schema-level. To specialize such synthesized loads for each particular design, the *load concretization* module exploits the abstraction function linking concrete designs to abstract models, but in reverse, and applied as a concretization function to a common test load for each concrete design. In the following sections, we describe the details of our approach.

4 Design Space Synthesis

This section presents our approach to synthesize design tradespaces for ORM using static metrics published in the database literature [28] [15]. The next two sections

then detail our approach to dynamic analysis of synthesized tradespaces, specifically synthesizing common loads and specializing abstract loads to the variant schemas in the design space, respectively.

We first introduce domain-specific languages (DSLs) that we developed within the Alloy language to let developers specify object and relational schemas in Alloy (§ 4.1). Next, we present formalizations of *object-relational mapping strategies* (§ 4.2) as well as verification of such semantic mapping rules (§ 4.3). We then use these formalizations to automate synthesis of design spaces of OR mappings (§ 4.4).

As an enabling technology, we chose Alloy [30] as a specification language and satisfaction engine. Alloy is a formal modeling language with a comprehensible syntax that stems from notations ubiquitous in object orientation, and semantics based on the first-order relational logic [30], making it an appropriate language for declarative specification of both application data object model and semantic mapping rules. More specifically, we chose it for three reasons. First, its logical and relational operators make Alloy an appropriate language for specifying object-relational mapping strategies. Second, its ability to compute solutions that satisfy complex constraints is useful as an automation mechanism. Third, Alloy has a rigorously defined semantics closely related to those of relational databases, thereby providing a sound formal basis for our approach.

The Alloy Analyzer is a constraint solver that supports automatic analysis of models written in Alloy. The analysis process is based on a translation of Alloy specifications into a Boolean formula in conjunctive normal form (CNF), which is then analyzed using off-the-shelf SAT solvers. With respect to the constraints in a given model, the Alloy Analyzer can be used either to find solutions satisfying them, or to generate counterexamples violating them. The Alloy Analyzer is a bounded checker, so a user-specified scope on the size of the domains needs to be specified. In the matter of the object-relational mappings, the scope states the number of elements of each top-level signature, such as Class, Attribute and Association. The analysis is thus performed through exhaustive search for satisfying instances within the specified scopes.

4.1 AlloyOM Domain-specific Language

To carry out the synthesis, we begin by developing domain-specific languages in Alloy that specify the source and target languages, for describing object and relational schemas, respectively. These specifications define element types, and how they are related and constrained to constitute valid expressions in these respective domains.

Listing 2 outlines the specification of the element types in the AlloyOM DSL. The DSL built-ins are defined as top-level Alloy signatures. A signature paragraph introduces a basic element type and a set of its relations, called *fields*, accompanied by the type of each field.

```
1  abstract sig Class{
2    attrSet: set Attribute,
3    id: set Attribute,
4    parent: lone Class,
5    isAbstract: one Bool
6  }
7
8  abstract sig Attribute{}
9
10 abstract sig Association{
11   src: one Class,
12   dst: one Class,
13   src_multiplicity: one Multiplicity_State,
14   dst_multiplicity: one Multiplicity_State
15 }
```

Listing 2: Partial specification of the AlloyOM DSL in Alloy.

The specification defines three main constructs as top-level signatures to model the basic AlloyOM constructs: `Class`, `Attribute` and `Association`. Note that these signatures are defined as *abstract*, meaning that they cannot have instance objects without explicitly extending them.

According to lines 2–5, the `Class` signature contains four fields: *id*, *parent*, *attrSet* and *isAbstract*. The `id` field within the `Class` signature (line 3) represents the identifier of the corresponding class. The inheritance relationship in the object model is represented by the `parent` relation in the AlloyOM DSL. Specifically, the inheritance relationship between two classes of c and p, where c inherits from p, for example, will represent by the expression of `parent = p` specified within the c class signature definition. The keyword ***lone*** specifies that an element is optional. In this specific case it indicates that the parent element is optional, and a Class may have one or no declared parent. The `attrSet` specifies the set of attributes as defined for the corresponding class in the object model. Finally, the `isAbstract` field denotes whether a given class is abstract or not; the keyword ***one***, used in its definition, states that every `Class` object is mapped to exactly one Bool object, representing its abstract state.

To describe each attribute within the AlloyOM DSL, one can define a signature that extends the corresponding data type signature. The AlloyOM DSL provides a set of predefined data types, such as Integer, Real, String and Bool, that extends the basic `Attribute` signature (line 8). Each `Association` signature contains four fields (as stated in lines 11–14): `src`, `dst`, `src_multiplicity` and `dst_multiplicity`. The first two fields specify the source and destination of the association, respectively. Association multiplicity defines the number of object instances that can be at each end of the association. The `src_multiplicity` and `dst_multiplicity` fields, thus, represent association multiplicities of source and destination classes respectively, and can have values of either `ONE` or `MANY`.

The code snippet in Listing 3 represents the diagram of Figure 1 delineating an object model for a simple

```
1  one sig Customer extends Class{}{
2    attrSet = customerID +customerName
3    id=customerID
4    isAbstract = No
5    no parent
6  }
7
8  one sig Order extends Class{}{
9    attrSet = orderID + orderValue
10   id = orderID
11   isAbstract = No
12   no parent
13 }
14
15 one sig CustomerOrderAssociation extends Association{}{
16   src = Customer
17   dst = Order
18   src_multiplicity = ONE
19   dst_multiplicity = MANY
20 }
21
22 one sig PreferredCustomer extends Class{}{
23   attrSet = discount
24   id = customerID
25   parent = Customer
26   isAbstract = No
27 }
```

Listing 3: Object model of Figure 1 formally modeled in our AlloyOM DSL.

customer-order example. The order class has two attributes of `orderID` and `orderValue`, which are assigned to the `attrSet` field of the `Order` class. The `id` field specifies the `orderID` as the identifier of this class. The last two lines of the `Order` signature specification denote that `Order` is not an abstract class and has no parent. Similarly, the code snippet of lines 8–13 represents `PreferredCustomer` signature definition. According to the diagram, the `PreferredCustomer` class inherits from the `Customer` class. The expression on line 11, thus, specifies `Customer` as the parent of the `PreferredCustomer` class.

To assist the users of Trademaker and to further make our tool chain more accessible, we have developed a front-end transformer that translates UML-based object models—simply can be modeled by UML modeling tools, such as ArgoUML[2]—to a formal representation in the AlloyOM language. Figure 4 shows a snapshot of the AlloyOM model transformer. Such a model transformer relieves the Trademaker's user of responsibility for developing formal specifications, while facilitating the use of the entire tradeoff analysis tool chain. The AlloyOM model transformer, along with more details on the AlloyOM language, is available from the project's website at http://jazz.cs.virginia.edu:8080/Trademaker/help.jsf.

A similarly defined Alloy-embedded relational schema specifies the *co-domain* of an object-relational map. A relation schema, in the relational model [22], is a set of attributes; a relational schema is thus a set of relation schemas; a relation, at the instance level, is a set of tuples over the attributes defined in the relation schema. Each relation schema has a primary key that may consist of one or more attributes of a

2. http://argouml.tigris.org/

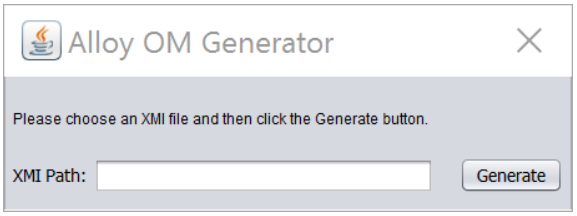

Fig. 4: A front-end transformer that translates UML-based object models to a formal representation in the AlloyOM language.

relation schema. A foreign key is a set of attributes of a schema used to reference relation instances (i.e., tuples) of another schema; foreign keys define referential constraints between relations.

4.2 Formalizing ORM Strategies as Semantic Mapping Rules

After specifying the abstractions involved in the source and destination domains, the next step is to express mapping rules as additional predicates that relate elements of the source domain to the constructs in the destination. These mapping rules, which basically provide a non-deterministic operational semantics to expressions in AlloyOM, are specified once as a semantic mapping predicate in relational logic. Combining these rules with a particular AlloyOM source model and finding all models of the resulting set of constraints reveals the set of corresponding design solutions.

To define such mapping rules, we rely on object-relational mapping strategies, informally defined in the literature [18], [28], [31], [35], and formalize them using relational logic. Specifically, to provide a basis for precise modeling of the space of mapping alternatives, we have formalized ORM strategies in an appropriate level of granularity.

To manage association relationships, we have formally specified three ORM strategies of *own association table*, *foreign key embedding* and *merging into single table* [35]. We have also specified three more ORM strategies for inheritance relationships: *class relation inheritance (CR)*, *concrete class relation inheritance (CCR)* and *single relation inheritance (SR)* [18]. Furthermore, as the aforementioned ORM strategies for inheritance relationships are just applicable to the whole inheritance hierarchies, we have specified extra predicates for more fine-grained strategies [35]: *Union Superclass*, *Joined Subclass* and *Union Subclass*, suitable to be applied to part of inheritance hierarchies, letting the developer design a detailed mapping specification using the combination of various ORM strategies. In fact, different fine-grained mapping strategies can be applied into different parts of a multi-level hierarchy. There are, however, some constraints that prevent all possible combinations. To make the idea concrete, in the rest of this section, we illustrate the semantics of these strategies that were first briefly introduced in our prior work [13], [14].

```
1  module ORMStrategies
2
3  pred unionSubclass[c:Class]{
4    // Assigning a table to each concrete class
5    (c.isAbstract = No) implies { one t: Table |
6      t.tAssociate = c and one t.tAssociate }
7    one c.~tAssociate
8    // relational fields corresponding to attributes of
         the associated class
9    (c.isAbstract = No) implies { all a: Attribute |
10     a in c.attrSet => oneAssocFieldClass[a, c] }
11   // relational fields corresponding to inherited
         attributes of the class
12   (c.isAbstract = No and some c.^parent) implies { all a
         : Attribute |
13     a in c.^parent.attrSet => oneAssocFieldClass[a, c] }
14   #c.~tAssociate.fields = # (c.*parent).attrSet
15   // Assigning the primary Key
16   (c.~tAssociate).primaryKey = (c.id.~fAssociate)
17   c.isAbstract = No implies oneAssocFieldClass[c.id, c]
18   // Assigning the foreign Key
19   no c.~tAssociate.foreignKey
20 }
```

Listing 4: A snippet of the Alloy model of the *union subclass* strategies for mapping classes within inheritance hierarchies.

```
22 pred joinedSubclass[c: Class] {
23   // Assigning a table to each concrete class
24   (c.isAbstract = No) implies { one t: Table |
25     t.tAssociate = c and one t.tAssociate }
26   one c.~tAssociate
27   // relational fields corresponding to non-inherited
         attributes of the associated class
28   some c.parent implies {
29     #c.~tAssociate.fields = #c.attrSet + 1 // for the
           foreign Key
30   } else {
31     #c.~tAssociate.fields = #c.attrSet
32   }
33   (c.isAbstract=No) implies { all a: Attribute |
34     a in c.attrSet => oneAssocFieldClass[a, c] }
35   // Assigning the primary Key
36   c.~tAssociate.primaryKey = c.id.~fAssociate
37   c.isAbstract = No implies oneAssocFieldClass[c.id, c]
38   // Assigning the foreign Key
39   some c.parent implies {
40     c.~tAssociate.foreignKey = c.parent.id.~fAssociate }
41   else { no c.~tAssociate.foreignKey }
42 }
```

Listing 5: A snippet of the Alloy model of the *joined subclass* strategies for mapping classes within inheritance hierarchies.

4.2.1 Mapping of inheritance relationships

Listings 4–5 show the semantics of fine-grained strategies for mapping individual classes within inheritance hierarchies. Each ORM strategy is represented by a separate Alloy predicate.

Union Subclass ORM strategy. The first Alloy predicate (Listings 4), parameterized by Class c, outlines the unionSubclass strategy (lines 3–20). This pattern implies mapping c into a separate relational table, should c be a concrete class. The tAssociate is a relation from a table to its corresponding class(es). Using the join operator, t.tAssociate thus states a set of all classes handled by the t table. The $\sim$ operator represents the transpose operation over a binary relation, which reverses the order of atoms within the relation. The statement of line 7, thus, using the multiplicity keyword one states that c is mapped to exactly one relational table. The strategy predicate then states, in lines 9–14, that the table encompasses relational fields corresponding to both attributes defined by c and all attributes inherited to this class. The in keyword represents the set inclusion operator. The operators '^' and '*' represent transitive closure and reflexive transitive closure, respectively. The '#' operator is the Alloy set cardinality operator. The expression c.^parent.attrSet (line 13) represents the set of all attributes inherited to c from its parents following one or more traversals along parent edges. As such, to retrieve object instances of this class, i.e., all objects whose most specific type is c, only one table needs to be accessed. This strategy thus implies no referential constraint over the mapped relations. As a concrete example, in the example shown in Figure 2c, a separate table is associated to the PreferredCustomer class, which is being mapped by the unionSubclass strategy. Note that the predicate specification relies on a separate predicate, oneAssocFieldClass (lines 65–69), that assigns a table field to a class attribute given as input.

Joined Subclass ORM strategy. The joinedSubclass strategy is then presented in Listing 5, lines 22–42. Using this strategy, the attributes defined by the class and the primary key attributes inherited from its super class are mapped to a separate table (lines 24–34). The table structure mapped using this pattern simply resembles the application object model, and the impact of changes in a class is limited only to the scope of the corresponding table, thus improving its maintainability. An example for application of the joinedSubclass strategy is shown in Figure 2a, that leads to a separate table for PreferredCustomer with a foreign key to its superclass (Customer) corresponding table. Different from the unionSubclass strategy, the relational table mapped using this pattern only includes attributes defined by the class c, rather than all its inherited attributes. Object instances are thus scattered over several rows in different relational tables. So to retrieve an object instance, it may require several joins that in turn may negatively affect the query performance [35].

Union Superclass ORM strategy. The statements shown in Listing 6 represent the Alloy predicate for the unionSuperclass mapping strategy, which implies mapping c into the same relational table as its super class(es) are mapped to. Such a table then stores object instances of c and all its transitive super classes, i.e., those classes reachable from c via the inheritance relationships (lines 47–48). Thus, when a class in an inheritance hierarchy is being mapped by the unionSuperclass strategy, all its super classes should also be mapped—using the same mapping pattern—into the same table the given class is mapped to. Moreover, a type indicator field is needed to distinguish the class type of each row in the table (lines 52–55). In the example of Figure 2b, a table is associated to both Customer and PreferredCustomer classes, which

```
pred unionSuperclass[c: Class] {
  // Assigning a table to each concrete class
  (c.isAbstract = No) => one Table <: c.~tAssociate
  // Class c and all its superclasses are mapped to a
       single table
  c.*parent.~tAssociate = c.~tAssociate
  // relational fields corresponding to both attributes
       of the associated class and its inherited
       attributes
  (c.isAbstract = No) implies { all a:Attribute |
    a in c.*parent.attrSet => oneAssocFieldClass[a, c] }
  // The table has an additional field distinguishing
       the type of record stored
  one f: Field| f.fAssociate in DType  and
    f in c.~tAssociate.fields and
    c.~tAssociate.fields = c.~tAssociate.foreignKey + c
        .~tAssociate.tAssociate.attrSet.~fAssociate + f
  // Assigning the primary Key
  c.~tAssociate.primaryKey = c.id.~fAssociate
  // Assigning the foreign Key
  (no a:Association| a.dst in c.*parent) =>{
    # (c.*parent.~tAssociate).foreignKey = #{ a:
        Association|
      a.dst in c.*parent and a.dst_multiplicity = MANY
          and no a.~tAssociate }
  }
}
// Assigning a relational field to a class attribute
pred oneAssocFieldClass[a: Attribute, c: Class] {
  one f: Field |
    f.fAssociate = a and
    f in c.~tAssociate.fields
}
```

Listing 6: A snippet of the Alloy model of the *union superclass* strategies for mapping classes within inheritance hierarchies.

are being mapped by the unionSuperclass strategy. This mapping pattern significantly reduces the number of tables in the relational schema, and no joins are needed to retrieve object instances of a single class. However, it may introduce NULL values into the database. Specifically, storing object instances of super classes into such a table mapped using the unionSuperclass strategy introduces NULL values for attributes defined just for a child class [28].

4.2.2 Mapping of Associations

There are several alternatives for the mapping of an association in the object model. We illustrate the semantics of the corresponding mapping strategies in the following. Note that since *n*-ary associations are typically transformed into binary ones in design models [35], our discussion in this section focuses on different possible mapping alternatives for binary relationships.

Own Association Table ORM strategy. The Alloy predicate of Listing 7, parameterized by Association asc, outlines the ownAssociationTable strategy. This pattern implies mapping each of the classes and the asc to separate relations. An example of this mapping strategy is shown in Figure 2a. The representation of the association in a separate table improves comprehensibility of the relational schema mapped using this pattern, and supports changes in cardinality of classes involved at either end of the association. Note that there are three types of association relationships based on the

```
pred ownAssociationTable[asc: Association] {
  // Assigning a table to the association and each of
       its ends
  one t: Table|  asc.src in t.tAssociate
  one t: Table|  asc.dst in t.tAssociate
  one t: Table|  t.tAssociate = asc
  one asc.~tAssociate
  one asc.src.~tAssociate
  one asc.dst.~tAssociate
  // relational fields corresponding to attributes of
       the associated classes
  assocEnd[asc.src]
  assocEnd[asc.dst]
  asc.~tAssociate.fields = asc.src.~tAssociate.
       primaryKey + asc.dst.~tAssociate.primaryKey
  #asc.~tAssociate.fields = 2
  // Assigning the primary Key
  asc.src.~tAssociate.primaryKey =
    asc.src.id.~fAssociate
  asc.dst.~tAssociate.primaryKey =
    asc.dst.id.~fAssociate
  asc.~tAssociate.primaryKey = asc.src.~tAssociate.
       primaryKey + asc.dst.~tAssociate.primaryKey
  // Assigning the foreign Key
  asc.~tAssociate.foreignKey = asc.src.id.~fAssociate +
       asc.dst.id.~fAssociate
  fKeysForMany[asc]
}

pred assocEnd[aEnd: Class] {
  all a: aEnd.attrSet {
    a !in Class => oneAssocField[a]
  }
  aEnd.~tAssociate.fields in
  aEnd.~tAssociate.foreignKey + aEnd.~tAssociate.
       tAssociate.attrSet.~fAssociate + DType.~fAssociate
}

pred oneAssocField[a: Attribute] {
  one f: Field {
    f.fAssociate = a
    one f.fAssociate
    one a.~fAssociate
    f in a.~attrSet.~tAssociate.fields
  }
}
```

Listing 7: A snippet of the Alloy predicate for the *own association table* strategy for mapping association relationships.

multiplicities of source and destination classes involved in such relationships. *One-to-one* relationships are those in which association multiplicity at each end is limited to one. In *one-to-many* relationships, participation of one end in the association is greater than one. Finally, in *many-to-many* relationships, association multiplicities at both ends are greater than one. The ownAssociationTable strategy is applicable to each of these three types. At the same time, should query performance be an important issue, it may not be a best mapping alternative.

Foreign Key Embedding ORM strategy. Listing 8 presents the Alloy predicate for the foreignKeyEmbedding ORM strategy. Using this mapping, a foreign key to the table, that corresponds to the class involved in the relationship with the multiplicity of one, would be embedded into the table corresponding to the other end of the association. A foreign key is a non-empty set of attributes of a relation which is used to point to tuples of another relation. Essentially, foreign keys specify referential constraints between relations. Figure 2b

```
110 pred foreignKeyEmbedding[asc: Association] {
111   // Assigning a table to  each end of the association
112   one t:Table| asc.dst in t.tAssociate
113   some asc.dst.~tAssociate
114   !((asc.src.~tAssociate = asc.src.~parent.~tAssociate)
115   or(asc.src.~tAssociate = asc.src.parent.~tAssociate))
116     implies one t:Table| t.tAssociate=asc.src
117   some asc.src.~tAssociate
118   // No table is assigned to the association
119   no t:Table| t.tAssociate = asc
120   no asc.~tAssociate
121   // relational fields corresponding to attributes of
        the associated classes
122   assocEnd[asc.src]
123   assocEnd[asc.dst]
124   // Assigning the primary Key
125   asc.src.~tAssociate.primaryKey =
126     asc.src.id.~fAssociate
127   asc.dst.~tAssociate.primaryKey =
128     asc.dst.id.~fAssociate
129   asc.src.~tAssociate.primaryKey in
130     asc.dst.~tAssociate.fields
131   asc.src.~tAssociate.primaryKey in
132     asc.dst.~tAssociate.foreignKey
133   // Assigning the foreign Key
134   fKeysForMany[asc]
135 }
```

Listing 8: A snippet of the Alloy predicate for the *foreignKey embedding* strategy for mapping association relationships.

```
136 pred mergingOneTable[asc: Association] {
137   //Both Classes and Associations would be merged into a
        sigle Table
138   one t: Table {
139     t.tAssociate = asc + asc.src + asc.dst
140     #t.tAssociate = 3
141   }
142   one asc.~tAssociate
143   // relational fields corresponding to attributes of
        the associated classes
144   all a: asc.(src + dst).attrSet {
145     a !in Class => (one f: Field {
146       f.fAssociate = a
147       f in a.~attrSet.~tAssociate.fields
148     } )
149   }
150   one f: Field {
151     f.fAssociate = asc.src
152     f in asc.~tAssociate.fields
153   }
154   one f: Field {
155     f.fAssociate = asc.dst
156     f in asc.~tAssociate.fields
157   }
158   #Field = #Field.fAssociate
159   // In one-to-one association relationships, the
        primary key of the table associated to the
        combination of Classes and their Association can
        be the primary key of either of classes, here it
        is assigned to the primary key of the src of the
        association
160   asc.~tAssociate.primaryKey = Field <: asc.src.id.~
        fAssociate
161 }
```

Listing 9: A snippet of the Alloy predicate for the *merging into one table* strategy for mapping association relationships.

shows an example application of this mapping strategy. This mapping pattern is applicable to the *one-to-one* and *one-to-many* relationships. Because association relationships in relational schema are maintained through the use of foreign keys, this mapping reduces the redundancies observed in relational tables mapped using *mergingOneTable*, discussed next. However, relational tables mapped using this pattern are not as maintainable as those mapped using the previous pattern, because, among other things, any cardinality changes to many-to-many are not easily integrated [35].

Merging into Single Table ORM strategy. The Alloy predicate of Listing 9 presents the mergingOneTable mapping strategy, in which both classes and the association are merged into a single relation. Applying this pattern would result in achieving better query performance, as no further joins are needed to retrieve object instances linked by *one-to-one* associations. It, however, may introduce NULL values into the table, especially when the mapping pattern applies to the other types of associations, such as *one-to-many* relationships.

4.3 Verifying Semantic Mapping Rules

Formalizing DSLs and mapping rules in an analyzable specification language not only enables automatic synthesis of transformed models, but also provides the basis to formally validate their correctness. By expressing essential properties of object-relational mappings required to be checked, we then use an automated relational logic analyzer to verify them. We specify such implications required to be checked as assertions. Assertions state a set of constraints intended to follow from specifications [30]. Correctness of mappings is then checked using a set of Alloy formula, i.e. assertions, that represent the expected relation between input object models and output relational schemas.

In the following, we illustrate the contents of two assertions, represented in Listing 10[3]. The first assertion (lines 1–3) states that no table should be associated solely to abstract classes, which is expected to hold in generated models according to the specified mapping rules. The next assertion is about the relational fields of each table. The point of this assertion is that a table may handle more than one class; it thus should contain relational fields associated with all those classes. The `tAssociate`, which first appeared in line 6 of the `tableFields` assertion, is a relation from a table to its corresponding class(es). Using the reverse join operator, $\sim$, the expression c.$\sim$tAssociate states the table associated to the class c, and then another join, .tAssociate, returns a set of all classes handled by that table. The expression of lines 6–7, thus, specifies that each table encompasses relational fields corresponding to attributes of all relevant classes. In some cases, the associated table also contains a separate field to indicate the most specific class for the object represented by each tuple. This type discriminator field is indicated as `DType` in the assertion under consideration.

The analyzer performs *scope-complete analysis* [30],

3. The complete model, including all assertions specifications, is available at the project's website.

```
1 assert noTableForAbstractClasses{
2   all c:Class| c.isAbstract=Yes => no t:Table| t.
        tAssociate = c
3 }
4
5 assert tableFields{
6   all c:Class| c.~tAssociate.fields in
7   c.~tAssociate.tAssociate.attrSet.~fAssociate +
8   c.~tAssociate.foreignKey + DType.~fAssociate
9 }
```

Listing 10: Two examples of assertions.

where each assertion is exhaustively checked against a huge set of model instances up to a certain bound. In other words, the analyzer is a bounded checker, guaranteeing the validity of assertions within a bounded instance space. We bound execution of assertion checking with the ultimate scope used for synthesis of transformed models, and thus expect the validity of assertions for all generated instances.

4.4 Design Space Exploration

In the previous section, we showed how analyzable specifications can be used to formalize ORM strategies. In this section, we tackle the other aspect that needs to be clarified: how one can apply a design space exploration approach to generate design spaces of OR mappings based on those specifications.

A design space is a set of possible design alternatives, and design space exploration (DSE) is the process of traversing the design space to determine particular design alternatives that not only satisfy various design constraints, but are also optimized in the presence of a set of additional objectives [38]. The process can be broken down into three key steps: (1) Modeling the space of mapping alternatives; (2) Evaluating each alternative by means of a set of metrics; (3) Traversing the space of alternatives, so characterized, to select one as the chose design.

4.4.1 Modeling the Space of Mapping Alternatives

For each application object model, due to a variety of different mapping options (cf. Sect. 4.2) available for each class, its attributes and associations, and its position in the inheritance hierarchy, there are several valid variants. To model the space of all mapping alternatives, we develop a generic mixed mapping specification based on fine-grained strategies formalized in previous section. This generic mixed mapping specification lets the automatic model finder choose for each element of the object model any of the relevant strategies, e.g. any of the fine-grained generalization mapping strategies for a given class within an inheritance hierarchy.

Applying such a loosely constrained mixed mapping strategy into the object model leads to a set of ORM specifications constituting the design space. While they all represent the same object model and are consistent with the rules implied by a given mixed mapping strategy, they exhibit different quality attributes. For example, how inheritance hierarchies are being mapped to relational models affects the required space for data storage and the required time for query execution.

We called this mapping strategy loosely constrained because it does not concretely specify the details of the mapping, such as applying, for example the *UnionSubclass* strategy to a specific class. An expert user, though, is able to define a more tightly constrained mixed mapping by means of the parameterized predicates Trademaker provides. The more detailed the mapping specifications, the narrower the outcome design space, and the less the required postprocessing search.

4.4.2 Static Analysis of Synthesized Designs

The choice of mapping strategy impacts key non-functional system properties. Response time performance, storage space and maintainability are among the set of quality attributes defined by the ISO/IEC 9126-1 standard that are influenced by the choice of OR mappings. To statically measure these attributes in an ORM design space, we use a set of metrics suggested by Holder et al. [28] and Baroni et al. [15]. The metrics are called Table Access for Type Identification (TATI), Number of Corresponding Table (NCT), Number of Corresponding Relational Fields (NCRF), Additional Null Value (ANV), Number of Involved Classes (NIC) and Referential Integrity Metric (RIM). In the following, we present three of these metrics for time and space performance.

Table Accesses for Type Identification (TATI)

Table Accesses for Type Identification (TATI) is a performance metric for polymorphic queries [28]. Intuitively, a high value for TATI(C) implies a longer execution time for a polymorphic query on C. According to the definition, given a class C, TATI(C) defines the number of different tables that correspond to C and all its subclasses. Our tools total up TATI values for each class as the overall TATI measure for each solution alternative.

Number of Corresponding Tables (NCT)

Number of Corresponding Tables (NCT) is a performance metric for insert and update queries. Intuitively, the performance of such queries mainly depends on the number of tables that hold data of the requested object. This metric, thus, specifies the number of tables that contain data necessary to assemble objects of a given class [28]. According to the definition, given a class C, NCT(C) equals to NCT of its direct super class, if C is mapped to the same table as its super class. Otherwise, if C is mapped to its own table, NCT(C) equals to NCT of its direct super class plus one. Finally, if C is a root class, NCT(C) equals to 1. Our tool computes totaled NCT values over classes as the NCT measure for each solution alternative.

Additional Null Value (ANV)

The Additional Null Value (ANV) metric specifies the storage space for null values when different classes are stored in a common table [28]. According to the definition, given a class C, ANV(C) equals to the number of non-inherited attributes of C multiplied by the number of other classes that are being mapped to the particular table to which C is mapped. Our tools present totals for ANV values over all classes as the ANV measures for each solution alternative.

To apply these metrics to synthesized solutions, we designed specific Alloy queries. Here we describe one for measuring the TATI metric. The others are evaluated similarly.

TATI(C) = #(C.(∼parent).∼tAssociate)*

Here the dot operator denotes a relational join. The Alloy ∼ operator represents the transpose operation over a binary relation, which reverses the order of atoms within the relation. Given the *tAssociate* (abstraction) relation that maps tables to their associated elements (i.e. Class or Association) within the object model, its transpose is the relation that maps each element to its associated table within the relational structure. The Alloy * operator represents the reflexive-transitive closure operation of a relation. Accordingly, the expression of "*C.*(∼parent)*" states a set of classes that have the class "C" as their ancestor in their inheritance hierarchy. The query expressions then, by using the Alloy set cardinality operator #, computes the TATI metric.

Our static metrics suite comprises six such static measures. The vector of these functions defines a 6-dimensional static analysis function applicable to Alloy-synthesized concrete designs (e.g., Figure 6). Our tools map this function over all elements of a synthesized design space to produce a tradeoff surface. The spider diagram, shown in Figure 5, illustrates one Pareto-optimal point on that surface for our example customer-order system. To display quality measures in one diagram, we normalized the values. Such diagrams can assist in conducting tradeoff analyses by making it easier to visualize and compare alternatives. According to the diagram, if the designer opts for performance, she may decide to use Sol. 5 instead of Sol. 4, as the latter has worse values for the TATI and NCT performance metrics.

4.4.3 *Exploring, Evaluating and Choosing*

The next step is to explore and prune the space of mapping alternatives according to quality measures. Trademaker partitions the space of satisfactory mixed mapping specifications into equivalence classes and selects at most a single candidate from each equivalence class for presenting to the end-user.

To partition the space, Trademaker evaluates each alternative with respect to previously described relevant metrics. So each equivalence class consists of all alternatives that exhibit the same characteristics. Specifically, two alternatives a_1 and a_2 are equivalent if *value*(a_1, m_i) = *value*(a_2, m_i) for all metrics (m_i). Because equivalent alternatives all satisfy the mapping constraints, we select one alternative in each equivalence class to find a choice alternative. Given that quality characteristics are usually conflicting, there is generally no single optimum solution but there are several pareto-optimal choices representing best trade-offs.

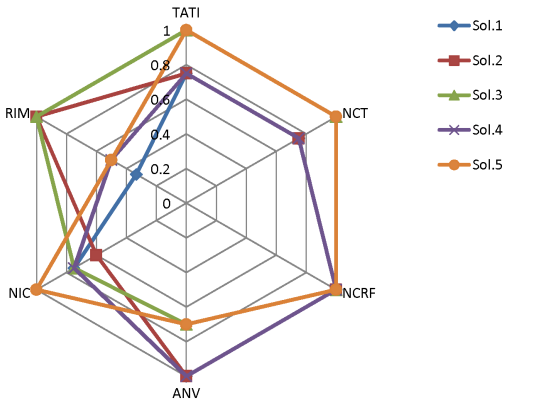

Fig. 5: Multi-dimensional *quality measures* for pareto-optimal solutions.

Having computed satisfying solutions, we then unparse these solutions from intermediate representation in relational logic into a desirable form, here SQL counterparts.

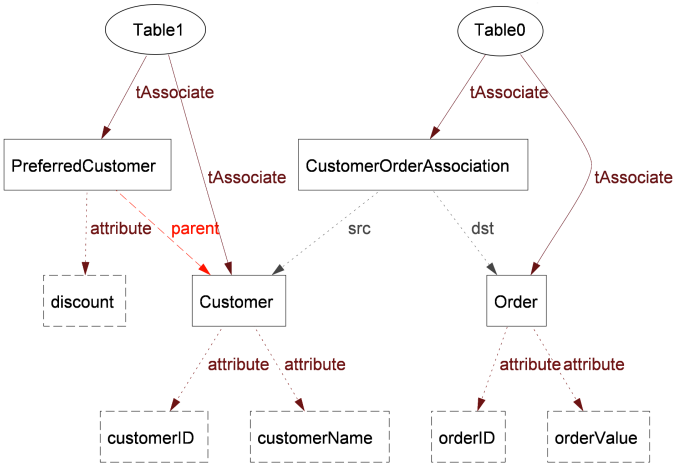

Fig. 6: OR mapping for customer-order example

Figure 6 presents a graphical depiction of an Alloy object encoding a synthesized OR mapping solution, where ovals represent tables, solid rectangles represent classes, and dotted boxes represent attributes. This solution is one of five Pareto-optimal solutions in the design space for our customer-order object model. The diagram is accurate but edited to omit some details for readability. In this diagram, Table1 is associated to Customer and PreferredCustomer classes, and Table0 is associated to both Order and CustomerOrderAssociation.

From this Alloy solution, our tools generate the SQL script of Listing 1. The script sets up a database with

the two tables: *Order*, with attributes *orderID*, *customerID* and *orderValue*; and *Customer*, with attributes, *customerID*, *customerName*, *discount*, and *DType*. Both *Customer* and *PreferredCustomer* objects are stored in this table under this particular mapping strategy, with the *DType* field distinguishing the type of record stored.

5 Abstract Load Synthesis

Our approach to synthesizing abstract loads starts with the automated transformation of a given Alloy-OM model into a related Alloy specification that we call a load model. We then use the Alloy Analyzer to synthesize abstract loads from this load model. Alloy solutions to the load model encode abstract object model data instances (*OM-instances*), which are what we take as synthesized loads with which to test synthesized designs. This section describes this functionality in more detail.

For each instance of *Class* and *Association* in the Alloy-OM model, our *model transformer* synthesizes a signature definition. When the class under consideration inherits from another class, the synthesized signature definition extends its parent signature definition. Given the specification of *Order* represented in Listing 3, the following code snippet represents its counterpart in a synthesized load model.

```
sig Order{
  orderValue: one Int,
  orderID: one Int
}
```

The *one* multiplicity constraints used in the declaration of elements' signatures within the Alloy-OM model (Listing 3) specify them as singleton signatures. While these constraints are required by the tradeoff space generator (e.g. to not generate multiple tables for a class in the model), they are unneeded for load generation, and thus omitted in the load model. The element attributes in the object model are also declared as fields of the corresponding load signature definition representing relations from the signature to the attribute *type*.

Finally, two sets of constraints are synthesized as *fact* paragraphs in the load model to guarantee both *referential integrity* of generated data as well as *uniqueness* of element identifiers with reference to the set of element instances to be generated. Referential constraints require every value of a particular attribute of an element instance to exist as a value of another attribute in a different element.

Consider the association relationship between Customer and Order classes from our running example (Figure 2). The code snippet of Listing 11 represents synthesized constraints in the load model for the customer-order association.

The expression of lines 1–3 states that if any two elements of type *CustomerOrderAssociation* have the same *orderID* and *customerID*, the elements are identical. This constraint rules out duplicate elements. The fact constraint of lines 5–8 states that for any orderID and customerID fields of a *CustomerOrderAssociation*, there are *Order* and *Customer* instances with the same *orderID* and *customerID*.

Applying the Alloy Analyzer to the derived load model yields the desired load in the form of object model data instances (OM-instances). Figure 7 depicts a generated OM-instance, which is essentially an Alloy solution object, where ovals represent associations, solid rectangles represent classes, and dotted boxes represent values. This solution represents two customers with customerID of 64 and 225, the latter a preferred customer with 10 percent discount, along with their orders. From many such solutions we derive an abstract (application-object-model-level, rather than concrete-database-schema-level) load with which to test the performance of many database instances.

Improving the efficiency of the load generator. One of the challenges we faced involved the scalability of this approach to load synthesis. A large number of solutions generated by the Alloy Analyzer were symmetric to previously generated instances, and thus did not contribute usefully to the load being generated. We explored a number of ways to improve efficiency of the load generator. The one that we found worked best is the iterative refinement of the load model by adding constraints that eliminate permutations of the already generated OM-instances. Without this improvement, it took 21 hours for Trademaker to generate test loads for one of our experiments. Given this approach, the time was reduced to about 2 hours—an order of magnitude speed up in the synthesis of test loads.

6 Abstract Load Concretization

The next challenge we discuss is to convert abstract load OM-instance objects into concrete SQL queries on a per-database basis. This is the task of specializing abstract load elements to the variant schemas presented by different solutions in the design space. Our *Alloy-to-SQL transformer* handles this task. To create SQL statements for a given database, the *Alloy-to-SQL transformer* requires an inverse OR mapping, namely the abstraction function that describes how a particular, concrete database schema implements the abstract object model.

Algorithm 1 outlines this transformation for *insert* queries, as it suffices to make our point. The approach supports the generation of select and update queries as well, which are important, of course, for comprehensive dynamic analysis.

The logic of the algorithm is as follows. Iterate over all elements in a given OM-instance (e.g., classes and associations) whose values can be populated into databases through insert statements. Look up the mapping to determine the table in which the element values should be stored. For each relational field in the associated table, if the OM-instance contains a value corresponding to that field, insert the value into the field. Otherwise, in the case that the field is a *DType*, insert the name of the element into the field. Finally, if the field is a foreignKey,

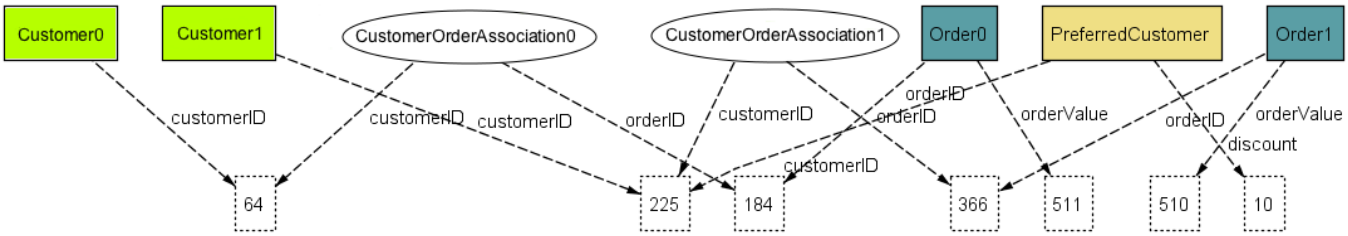

Fig. 7: An example of OM-instance.

```
fact {
  all o1,o2:CustomerOrderAssociation|o1.orderID = o2.
      orderID and o1.customerID = o2.customerID => o1=o2
}

fact {
  all o:CustomerOrderAssociation| one c:Order| o.orderID =
      c.orderID
  all o:CustomerOrderAssociation| one c:Customer| o.
      customerID = c.customerID
}
```

Listing 11: Part of the load model specification generated for customer-order association.

```
INSERT INTO 'Customer' ('customerID','DTYPE') VALUES (64,'
    Customer');
INSERT INTO 'Customer' ('customerID','DTYPE') VALUES (225,'
    PreferredCustomer');
INSERT INTO 'Order' ('orderID','orderValue','customerID')
    VALUES (184,511,64);
INSERT INTO 'Order' ('orderID','orderValue','customerID')
    VALUES (366,510,225);
```

Listing 12: Generated SQL insert statements from OM-instance of Figure 7 for implementation mapping of Figure 6.

Algorithm 1: Generate SQL Insert Statements

```
Input: omi: OM-instance, map: OR mapping
Output: A set of SQL insert statements
for element in omi do
    T = map.TableAssociat(element);
    F = T.fields;
    for field in F do
        value = getValueFromOMI(field);
        if value != null then
            add "field = value" into statements
        end
        else
            if field == "DType" then
                value = element.name;
            end
            if isForeignKey(field) then
                attr =
                 findAttributeFromAssociation(field);
                value = getValueFromOMI(attr);
            end
            add "field = value" into statements
        end
    end
end
```

find the associated attribute from a relevant association in the given OM-instance, and insert its value into the field.

Consider the database alternative for our running example, in which we store the customer-order association data into the order table (Figure 2b). In that case, the field of *customerID* in the *Order* table is a foreignKey, and its values come from the associated customerOrderAssociation element.

Listing 12 represents the set of SQL insert statements generated from the OM-instance of Figure 7 according to the mapping of Figure 6. The first two generated statements define insert queries to store instances of Customer and PreferredCustomer into the Customer table along with appropriate *DType* values for each one. The next two statements then store instances of Order and CustomerOrderAssociation into the Order table.

7 Tool Implementation

We have implemented our approach in a tool called Trademaker, which is freely available[4]. Trademaker is a web-accessible tool that implements our ORM synthesis and analysis approach. It supports automated, specification-driven synthesis of ORM design spaces and static analysis using the aforementioned metrics. It provides a web interface, user account and job management (job submission, asynchronous execution, status reporting, persistence), computation and presentation of Pareto-optimal subsets of synthesized designs under the given metrics, and synthesis of SQL databases for selected designs.

Figure 8 presents a screenshot of a Trademaker run. Rows present Pareto-optimal designs, and columns, analysis results. It takes an object model as input, expressed in the Alloy-OM DSL (cf., section 4.1), uses the Alloy Analyzer to exhaustively enumerate satisfying solutions, applies static analysis functions to each design, filters them for Pareto optimality, presents the

4. Research artifacts and experimental data are available at http://www.jazz.cs.virginia.edu:8080/Trademaker

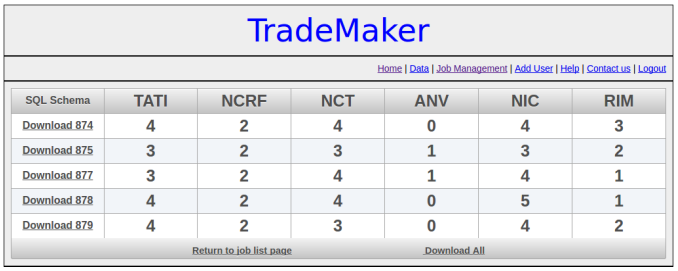

Fig. 8: A view of our tool to provide decision-makers with Pareto-optimal OR mapping solutions based on static analysis results; columns and rows represent metrics and solution alternatives, respectively.

results—design solutions and property estimates in six dimensions—and delivers usable MySQL scripts to instantiate selected designs.

8 Experimental Validity Test of Static Prediction Rules

As an experimental test of our approach to specification-driven, automated dynamic analysis of non-functional property tradeoffs across design spaces, we apply the approach to test the validity of the static predictors of database performance. We formulate, test, and provide experimental data in support of three driving hypotheses:

- H1: The ordering of alternatives predicted by the static metrics predicts that of the dynamic analysis results
- H2: The relative magnitudes of static measures of alternatives predict those of the dynamic analysis results
- H3: Dynamic analysis using scale-limited synthesized loads predicts performance under much larger loads

This section summarizes the design and execution of our experiment, the data we collected, its interpretation, and our results.

8.1 Subject Systems

We synthesized design spaces and compared static predictions with dynamic results for six subject systems, selected from different sources and of a variety of different domains, ranging from our research lab projects to applications adopted from the database literature and open-source software communities. The selected experimental subjects are representative of a large class of useful applications at a scale matched to the state-of-the-art synthesis techniques. Their object models vary in terms of size and complexity, and contain multiple association and inheritance relationships, which induce many possible choices of object-model to relational schema mappings. Table 1 shows characteristics of these systems in terms of the number of classes, associations, and inheritance relationships.

The first is the object model of an E-commerce system adopted from Lau and Czarnecki [34]. It represents a common architecture for open-source and commercial E-commerce systems. It has 15 classes connected by 9 associations with 7 inheritance relationships. The second and third object models are for systems we are developing in our lab. Decider is another system to support design space exploration. Its object model has 10 Classes, 11 Associations, and 5 inheritance relationships. The third object model is for a system, CSOS, a kind of cyber-social operating system meant to help coordinate people and tasks. In scale, it has 14 Classes, 4 Associations, and 6 inheritance relationships. The fourth and fifth object models are from two open source applications. We obtained their object models by reverse engineering of their database schemas. WordPress is an open source blog system [3]. Its object model has 13 classes connected by 10 associations with 8 inheritance relationships. Moodle is a learning management system [1], which is widely used in colleges and universities. It has 12 classes connected by 8 associations and consists of 4 inheritance relationships. We also analyzed an extended version of our customer-order example.

TABLE 1: Subject systems.

Subject System	# Classes	# Associations	# Inheritance relationships
Decider	10	11	5
E-commerce	15	9	7
CSOS	14	4	6
WordPress	13	10	8
Moodle	12	8	4
Cust-order (ext)	4	2	1

8.2 Planning and Execution

Our experimental procedure involved the synthesis of both design spaces of database alternatives and several abstract loads in a variety of sizes for each subject system. Given the synthesized schemas, we created a database for each alternative. We then populated generated data into databases, and ran concrete queries over those databases. We measured and collected the number of concrete queries generated from abstract loads for each database alternative, query execution time, as well as the size of each database.

We used an ordinary PC with an Intel Core i7 3.40 Ghz processor and 6 GB of main memory, with SAT4J as our SAT solver. Database queries were performed on a MySQL 5.5.30 database management system (DBMS), installed on a machine equipped with an AMD Opteron 6134 800 Mhz processor and 64GB memory. Data and statistical information are available at http://jazz.cs.virginia.edu:8080/Trademaker/data.

Table 2 summarizes the generated solution space for each subject system. There is one row for each system. The columns indicate the total number of solutions, the number of static equivalence classes where equivalence is determined by equality of static analysis results, and

TABLE 2: Design space sizes for subject systems.

Subj. Sys.	Solutions	Eq.Classes	Pareto Sols.
Decider	386	154	12
E-commerce	846	360	16
CSOS	278	121	21
WordPress	924	276	13
Moodle	586	144	12
Cust-order (ext)	28	14	10

TABLE 3: Part of the generated data sets for the E-commerce experiment; the second row shows abstract loads generated for the E-commerce system within each data set; each cell in the other rows corresponds to the size of generated concrete load (in terms of the number of select/insert statements) for the database alternative and data set given on the axes.

E-commerce	Dataset 1	Dataset 2	Dataset 3
abstract load	862	2,576	164,813
Sol.19	456/678	13,698/20,172	2,471,700/3,100,350
Sol.121	320/540	9,770/16,244	1,647,800/2,276,450
Sol.264	397/618	12,073/18,547	2,142,140/2,770,790
Sol.348	379/600	11,395/17,869	1,977,360/2,606,010

the number of Pareto-optimal solutions under the given static metrics.

We investigated and compared two different methods for generation of data sets. The first method generated data using our formal synthesis methods. For the second, we hand-developed a load generator for generating large loads that nevertheless respect the constraints in our object models (e.g., referential constraints between elements).

Three data sets were developed for each subject system to support the task of evaluating the static metrics.

Dataset 1. This data set is generated using the Alloy-based data generator, where the maximum bit-width for integers is restricted to 5. This leads to the generation of a small data set for our experiments.

Dataset 2. This data set is generated using our Alloy-based data generator. The maximum bit-width for integers is restricted to 10, which leads to the generation of a larger data set compared with the former data set.

Dataset 3. As with many formal techniques, the complexity of constraint satisfaction restricts the size of models that can practically be analyzed and synthesized [23], [32]. For experimental purposes, we hand-implemented a more scalable data generator. It does not generate queries directly, but rather replaces the constraint solver for synthesis of abstract loads. Having synthesized larger abstract loads in the form of OM-instances, using the mechanisms already used in the Alloy-based data generator (cf. Section 5), the generator then transforms abstract loads into sets of concrete queries targeting diverse implementation alternatives.

Table 3 presents the size of generated data sets for some of the solution alternatives for the E-commerce system. Observe that the sizes of concrete queries—in the form of select/insert statements—refined from common abstract loads are different in various solution alternatives, depending on the way that each implementation mapping alternative refines an abstract object model into the concrete representation in relational structure (cf. Section 6).

Table 4 presents the time that it takes to execute the generated concrete loads for the same solution alternatives, as shown in Table 3, for the E-commerce system. We have handled uncontrollable factors in our experiments by repeating each experiment 10 times and computing the average execution time. According to the table, the way that each solution alternative refines the abstract data loads into the concrete SQL statements directly affects the the time that it takes to execute their corresponding concrete loads, and thus influences the performance of the design solution.

In the rest of this section, we present the experimental data to address the three hypotheses driving our research.

8.3 Results for Hypothesis H1 (Order)

To test the predictive accuracy of our static metrics, we compared its predictions against the results of our dynamic analysis. To evaluate our first hypothesis—whether the relative order of implementation alternatives is predicted by static metrics—we compute Spearman correlation coefficients, an appropriate correlation statistic for order-based consistency analysis. It measures the degree of consistency between two ordinal variables [41]. A correlation of 1 indicates perfect correlation, while 0 indicates no meaningful correlation. Negative numbers indicate negative correlations.

Table 5 summarizes correlation coefficients between static metrics and dynamic measures. The data show reasonably strong but somewhat inconsistent positive correlations between statically predicted and actual runtime performance for TATI (average correlation of 0.90) and NCT (average correlation of 0.88). These metrics appear moderately to strongly predictive of the relative ordering databases run-time performance, at least for the kinds of loads employed in our experiments.

The performance of the ANV predictor varies across the subject systems. ANV predicts well in the E-commerce, Decider, and Moodle experiments and moderately in the CSOS data, but weakly in the WordPress and customer-order-extended sets. Moreover, the results

TABLE 4: The running time (in seconds) to execute the generated concrete loads (select/insert SQL statement queries) for the solution alternatives shown in Table 3, for the E-commerce system. To account for the effects of uncontrollable factors in our experiments, we ran each set of data 10 times, and computed the average.

E-commerce	Dataset 1	Dataset 2	Dataset 3
Sol.19	0.090/0.313	2.584/5.802	763.920/401.420
Sol.121	0.069/0.255	1.921/4.744	555.079/305.316
Sol.264	0.084/0.276	2.251/5.318	694.825/346.232
Sol.348	0.083/0.272	2.174/5.110	623.970/329.959

show negative correlation between the ANV metric and database size. As the number of null values increases, size decreases. This observation for the ANV metric is in direct contrast to what is predicted. One possible reason is that when the ANV metric increases, the number of tables for the database solution under consideration decreases. Assuming that the database system efficiently stores null values, the database size would reduce.

To further evaluate predictive accuracy of static metrics, we consider the case in which designers use each static metric as a two-class classifier. We, thus, measure precision, recall and F-measure as follows:

Precision is the percentage of those alternatives predicted by a given metric as more preferable in terms of a given quality attribute that were also classified as more preferable by the actual analysis: $\frac{TP}{TP+FP}$

Recall is the percentage of alternatives classified more preferable by the actual analysis that were also predicted as more preferable by the a given metric: $\frac{TP}{TP+FN}$

F-measure is the harmonic mean of precision and recall: $\frac{2*Precision*Recall}{Precision+Recall}$

where TP (true positive), FP (false positive), and FN (false negative) represent the number of solution alternatives that are truly predicted as preferable, falsely predicted as preferable, and missed, respectively.

While the static metrics output predictions of quality characteristics as natural numbers, our actual dynamic analysis of query execution performance and required storage space are in terms of seconds and bytes, respectively. To classify an alternative as *preferable*, we thus use median for each set of result values as a threshold. We measure evaluation metrics for each subject system with respect to three data sets.

Table 6 summarizes the results of our experiments to evaluate the accuracy of static metric predictors as two-class classifiers. The average precision, recall and F-measure are depicted in Figure 9. The results show the accuracy of the TATI and NCT metrics in classifying implementation alternatives in terms of their run-time performance. The average precision and recall for all four experiments are about 90%, showing a low rate of both false positives and false negatives. The ANV metric, however, achieves an average under 30% in all evaluation metrics.

The experimental data thus suggests that, under the generated abstract loads, the relative order of implementation alternatives predicted by static metrics of TATI and NCT is indicative of their comparative preference in actual runtime performance, but this is not the case for ANV as a static predictor of storage space.

TABLE 5: Correlation coefficients between the relative order of solution alternatives predicted by static metrics and those observed from actual runtime measures.

		TATI	*NCT*	*ANV*
Decider	Dataset1	0.93	0.93	-0.98
	Dataset2	0.96	0.96	-0.92
	Dataset3	0.95	0.92	-0.91
E-commerce	Dataset1	0.97	0.96	-0.95
	Dataset2	0.98	0.95	-0.95
	Dataset3	0.97	0.94	-0.95
CSOS	Dataset1	0.83	0.76	-0.97
	Dataset2	0.57	0.62	-0.79
	Dataset3	0.58	0.74	-0.55
WordPress	Dataset1	0.95	0.97	-0.25
	Dataset2	0.96	0.97	-0.31
	Dataset3	0.96	0.97	-0.26
Moodle	Dataset1	0.95	0.90	-0.98
	Dataset2	0.99	0.96	-0.95
	Dataset3	0.99	0.92	-0.98
Cust-order(ext)	Dataset1	0.89	0.92	-0.51
	Dataset2	0.74	0.53	-0.44
	Dataset3	0.66	0.82	-0.56

TABLE 6: Experimental results of evaluating OR metrics as two-class classifiers.

		TATI			*NCT*			*ANV*		
		Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
Decider	DS1	1	0.75	0.86	0.83	0.83	0.83	0	0	0
	DS2	0.83	0.83	0.83	1	1	1	0.17	0.17	0.17
	DS3	1	1	1	0.83	0.83	0.83	0.17	0.17	0.17
E-commerce	DS1	1	1	1	1	1	1	0	0	0
	DS2	1	1	1	1	1	1	0	0	0
	DS3	0.9	0.9	0.9	1	1	1	0	0	0
CSOS	DS1	0.75	0.82	0.78	0.75	0.82	0.78	0.36	0.33	0.35
	DS2	0.83	1	0.91	0.67	0.80	0.73	0.36	0.33	0.35
	DS3	0.83	1	0.91	0.83	1	0.91	0.36	0.33	0.35
WordPress	DS1	1	1	1	1	1	1	0.14	0.14	0.14
	DS2	1	1	1	1	1	1	0.14	0.14	0.14
	DS3	1	1	1	1	1	1	0.25	0.29	0.27
Moodle	DS1	1	0.75	0.86	0.88	1	0.93	0.17	0.17	0.17
	DS2	1	1	1	0.75	1	0.86	0.17	0.17	0.17
	DS3	1	1	1	0.75	1	0.86	0	0	0
Cust-order (ext)	DS1	1	1	1	1	1	1	0.43	0.60	0.50
	DS2	0.83	1	0.91	0.67	0.80	0.73	0.57	0.67	0.62
	DS3	0.83	1	0.91	0.83	1	0.91	0.43	0.60	0.50

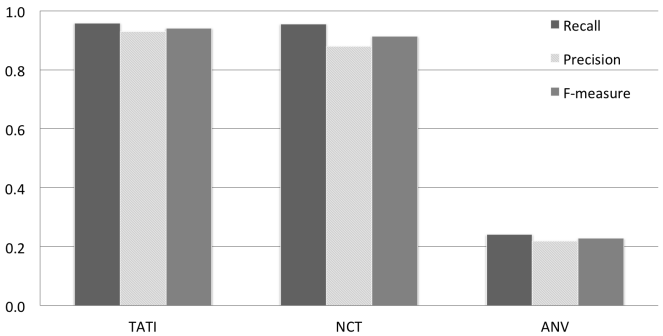

Fig. 9: Bar plot of the average precision, recall and F-measure for considering static metrics as two-class classifiers.

8.4 Results for Hypothesis H2 (Magnitudes)

To address the second hypothesis—relative magnitude of static predictions matter—we employ a coefficient of determination denoted R^2, as a metric for how well actual outcomes are predicted by the static metrics. Figure 10 plots the results. For brevity, only results from *Dataset 3* are presented; other data sets give similar results.

The performance of the predictors varies widely across systems and predictors. TATI and NCT are predictive

of performance for four of the systems, i.e., the E-commerce, Decider, WordPress, and Moodle systems, but relatively poor predictors for CSOS. TATI performs poorly in the customer-order-extended data. ANV predicts size in the E-commerce and Moodle experiments, and moderately in the Decider and WordPress data, but inconsistently and weakly in the CSOS data and not at all in the customer-order-extended set.

We interpret this data as suggesting that the relative magnitudes of static metrics for various solution alternatives are not reliably indicative of the relative magnitudes of actual performance, and that ANV is a poor indicator of the storage space. One is advised to use such static metrics with caution. While TATI and NCT metrics predict the relative order of solution alternatives with high confidence, the difference in predicted values of two alternatives is not a good indicator of their actual run-time difference.

8.5 Results for H3 (Small vs. Large Loads)

To address the third hypothesis—that small, formally synthesized loads predict the outcomes of much larger loads—we employ the Pearson product-moment correlation statistic. Pearson measures the degree of linear dependence between two variables, not necessarily ordinal, as opposed to the Spearman test. A correlation of 1 represents perfect correlation, and 0, no meaningful correlation.

We summarize correlation coefficients between experimental results obtained from smaller data sets of 1 and 2 and that of the large *Dataset 3* in Figure 11. The average Pearson correlation coefficient between $Dataset3$ and the first and second data sets are 88% and 94%, respectively. These data lend support to the proposition that smaller-scale test sets produced by specification-driven synthesis can provide valid predictions of performance under larger, more realistic loads.

9 Discussion and Limitations

The overarching problem this work addresses is interesting and important: the need for improved science and technologies to support decision making in complex and poorly understood tradeoff spaces, particularly involving tradeoffs among non-functional properties, also sometimes called *ilities*. We need languages in which to specify design spaces, techniques for synthesizing and analyzing design spaces, mechanisms for mapping static and dynamic analysis functions across design spaces, techniques for validating such metrics, and tools that enable engineers to use the science to improve real engineering practice. We also need measures for *ilities* that are important but hard to measure today: for evolvability, some dependability properties, affordability of construction, and more.

Trademaker takes an important step towards this overarching objective in the context of object-relational database mappings, but we envision the ideas set forth in this research to find a broader application in other computing domains as well. It suggests the possibility of useful formal languages for specifying design spaces in support of formal synthesis of both designs and comparative analysis loads. We showed that specialization of common loads is enabled by access to abstraction functions from concrete to abstract designs, which can be embedded in the results of design synthesis. Concretization functions proved useful not only for scale-limited, formally synthesized loads, but for concretizing abstract loads produced by other means.

Our experimental analysis using Trademaker to test the validity of static predictors of database performance based on published but not validated metrics indicates that two of the metrics appear to produce meaningful signals, while the third appears not useful. The data also indicate a need for caution in relying on the static metrics. Their predictive accuracy, even in the "good" cases, varied across application models. That said, we can now provide automated dynamic analysis as a fallback. We are integrating support for invoking such automated analysis into our Trademaker tool. Trademaker itself has real potential utility for object-relational mapping and partial application synthesis; but its greater significance is as a demonstration of our research results and a testbed for further research on formal tradespace modeling and analysis.

There are of course limitations in our approach and in this work. We mention those most relevant to a proper evaluation of this effort. First, the static metrics we evaluated *sum* the values of published metrics over the elements of each design alternative. We thus extended the original metrics and our statistical results should technically be read as pertaining to these extensions of the original measures.

Second, while our synthesis mechanisms are implemented and working, our infrastructure for running synthesized concrete loads against synthesized designs still relies on some manual processing. Our statistical data were thus derived by dynamic analyses of certain *subsets* of our synthesized designs. We selected the subsets deemed Pareto-optimal by the static metrics. As our infrastructure matures, we will conduct whole-space dynamic analyses, which we expect to produce results consistent with the basic result presented here. We are on a path to support automated whole-space dynamic analysis through Trademaker. The work reported in this paper did nevertheless involve the synthesis and dynamic analysis of over 300 database alternatives.

Third, our experiments to date tested our hypotheses for "random" loads of varying sizes. Real applications will generally produce non-random loads. Whether the static metrics we tested are predictive for large, real applications remains unclear. On the other hand, we offer dynamic analysis at scale as an alternative to static metrics. The proposed Trademaker tool suite further supports reasonable extension for new types of test load generators. In fact, separating abstract load synthesis

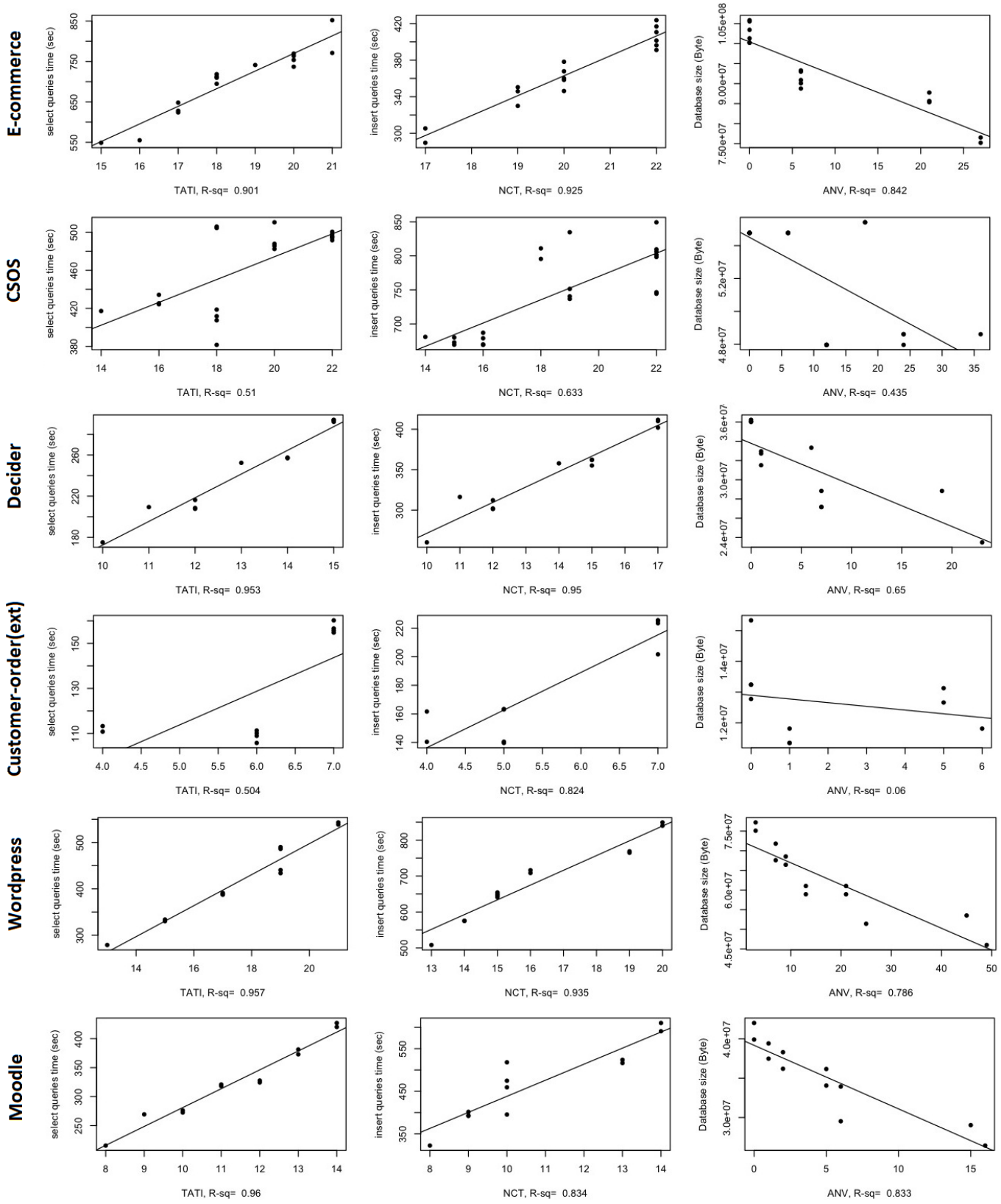

Fig. 10: Correlation between static metrics and actual run-time measure; columns represent scatter plots of observed values across systems versus predicted values by TATI, NCT and ANV metrics from left to right, respectively; R^2 correlation coefficient is shown at the bottom of each plot.

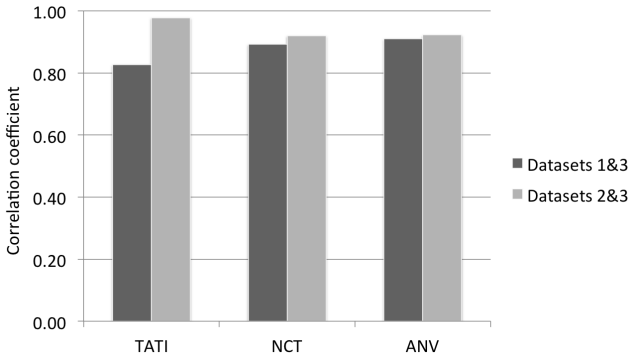

Fig. 11: Summary of Pearson correlation coefficients between experimental results obtained from smaller data sets and that of the large $Dataset3$.

from load concretization into variant designs, as proposed in the Trademaker architecture (cf. Section 3), enables extension/revision of each, independent of the other, thus supports the necessary extensions to enrich the tradeoff analysis environment. We envision a future in which some systems run many design variants in parallel, perhaps with small but representative loads abstracted from real loads on live systems, to detect conditions in which dynamic switching to new implementation strategies should be considered.

Finally, there is the issue of scalability. Using Alloy as a constraint solver entails scalability constraints. We can handle object models with tens of classes. Industrial databases often involve thousands of classes. It is unlikely that our current implementation technology will work at that scale. For now, it does have real potential as an aid to smaller-scale system development. That we can present an object model for a realistic web service, synthesize a broad space of ORM strategies, select one based on tradeoff analysis, automatically obtain an SQL-database setup script, provide it Java EE, and have much of an enterprise-type application up and running with little effort is exciting, even if it does not address (yet) the most demanding needs of industry.

10 Related Work

We can identify in the literature a large body of research related to ours. Here, we provide an overview of the most notable research and examine them in the light of our research.

Formal derivation of database implementations. A number of researchers have proposed formal approaches for deriving database-centric implementations from high-level specifications [19], [33]. Alchemy refines Alloy specifications into PLT Scheme implementations with a special focus on persistent databases [33]. Along the same line, Cunha and Pacheco proposed an approach that translates a subset of Alloy into the corresponding relational database operations [19]. Both Alchemy and Cunha and Pacheco's approach refine the specification into a single implementation, whereas Trademaker generates *spaces* of possible database design alternatives. While these research efforts share with ours the emphasis on using formal methods, our work differs fundamentally in its emphasis on the generation of *spaces* of implementation alternatives, not just point solutions.

Object-relational mapping. A large body of work has focused on object-relational mapping approaches to the object-relational impedance mismatch problem [18], [29], [31], [35]. Philippi [35] categorized the mapping strategies in a set of pre-defined quality trade-off levels, which are used to develop a model driven approach for the generation of OR mappings. This work similar to many other work we studied derives a single design solution from input specifications. Moreover, they did not apply static metrics, nor dynamic analysis to measure the effectiveness of design alternatives. Our technique is inspired in part by the work of Cabibbo and Carosi [18], discussed more complex mapping strategies for inheritance hierarchies, in which various strategies can be applied independently to parts of a multi-level hierarchy. Our approach is novel in having formalized ORM strategies previously informally described in some of these research efforts, thereby enabling automatic generation of OR mappings for each application object model.

Drago et al. [21] considered OR mapping strategies as a variation points in their work on feedback provisioning. They extended the QVT-relations language with annotations for describing design variation points, and provided a feedback-driven backtracking capability to enable engineers to explore the design space. While this work is concerned with the performance implications of choices of per-inheritance-hierarchy OR mapping strategies, it does not attack the problem that we address, the automation of dynamic analysis through synthesis of design spaces and fair loads for comparative dynamic analysis.

The other relevant thrust of research has focused on mapping UML models enriched with OCL invariants into relational structures and constraints. Among others, Heidenreich et al. [27] developed a model-driven framework to map object models represented in UML/OCL into declarative query languages, such as SQL and XQuery. While Heidenreich et al.'s approach concentrates on mapping OCL invariants into an implementation-level language to enforce semantical data integrity at the implementation level, Trademaker automatically generates database schemas mainly based on structural constraints. Badawy and Richta [6] provided some rules guiding derivation of declarative constraints and triggers from OCL specifications. These two research work are complementary. Extending the same line, Al-Jumaily et al. [26] developed a model-driven tool transforming the OCL constraints into SQL triggers. Demuth et al. [20] also discussed a number of different approaches to implement OCL-to-SQL mapping, and developed a tool that transforms each OCL invariant

into a separate SQL view definition. Different from these research efforts transforming an object model to a single counterpart in relational structures, Trademaker generates tradeoff spaces of object-relational mappings with focus on structural mapping alternatives, rather than transformation of integrity constraints.

Generating test loads. Numerous techniques have been developed for generating testing loads [16], [17], [32], [36], [43], including the generation of realistic loads for TPC benchmarks [2]. Among others, Khalek et al. proposed a query-aware test generation technique, called ADUSA [32]. Given a database schema and an SQL query as inputs, ADUSA then exhaustively generates non-isomorphic test databases. Similar to many other techniques including ours, ADUSA uses a constraint solver as a test generation engine. However, our work is different in that no prior technique generates common test loads over spaces of alternative schemas. Doing this requires enforcement of abstract design constraints as well as constraints implied by concretization mappings for each alternative. Trademaker, to our knowledge, is the first tool with this capability.

Constraint solving for analysis and synthesis. Finally, constraint solving for synthesis and analysis has increasingly been used in a variety of domains [5], [7]–[12], [25], [39], [40], [44]. Sketching [5] is a synthesis technique in which programmers partially define the control structure of the program with holes, leaving the details unspecified. This technique uses an unoptimized program as correctness specification. Given these partial programs along with correctness specification as inputs, a synthesizer — developed upon a SAT-based constraint solver — is then used to complete the low-level details to complete the sketch by ensuring that no assertions are violated for any inputs. This work shares with ours the common insight on both using partial specifications and synthesis based on constraint solving. However, our work focuses on automating mapping from practically meaningful abstract application object models to database design structures.

Along the same line, Srivastava et al. [39] developed a proof-theoretic synthesis, in which the user provides relations between inputs and outputs of a program in the form of logical specifications, specifications of the program control structure as a looping template, a set of program expressions, and allowed stack space for the program to be synthesized. It then generates a constraint system such that solutions to that set of constraints lead to the specified program. They have shown feasibility of their approach by synthesizing program implementations for several algorithms form logical specifications.

Different from the aforementioned techniques that mainly focus on low-level details of programs, Trademaker tackles the automated tradeoff space analysis of ORMs, by synthesizing spaces of design alternatives and common loads over such spaces. It thus relieves the tedium and errors associated with their manual development. To the best of our knowledge, Trademaker is the first formally precise technique for automated synthesis and dynamic analysis of tradeoff spaces for object-relational mapping.

11 Conclusion

This paper makes several contributions to the science and engineering of software-intensive systems: a novel approach for formal, automated tradeoff analysis derived by synthesis from relational logic models; a principled approach to load concretization for specializing common loads to large numbers of variant implementations; and Trademaker, an accessible and functional tool enabling tradeoff analysis in large design spaces for the particular domain of object-relational mapping, and a testbed for ongoing research of the kind reported in this paper. This paper also contributes to our broader research program, which is increasingly focused on specifying, validating, realizing, and certifying acceptable tradeoffs among non-functional properties, which remains a research challenge of the first order.

12 Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant Number CMMI 1400294. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We thank Michele Claibourn and Clay Ford of UVa StatLab for their consulting and statistical assistance.

References

[1] Moodle. http://docs.moodle.org/dev/Grades#Database_structures.
[2] TPC benchmarks. http://www.tpc.org.
[3] WordPress. http://codex.wordpress.org/Database_Description/3.3.
[4] R. Al-Ekram, R. Holt, C. Hobbs, and S. Sim. Automating service quality with tomcad (tradeoff model with capacity and demand). In *Proceedings of the 2007 Workshop on Automating Service Quality: Held at the International Conference on Automated Software Engineering (ASE)*, WRASQ '07, pages 4–9, New York, NY, USA, 2007. ACM.
[5] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.
[6] M. Badawy and K. Richta. Deriving triggers from UML/OCL specification. In *Information Systems Development*, pages 305–315. Springer US, Jan. 2002.
[7] H. Bagheri, E. Kang, S. Malek, and D. Jackson. Detection of design flaws in the android permission protocol through bounded verification. In *FM 2015: Formal Methods*, volume 9109 of *Lecture Notes in Computer Science*, pages 73–89. Springer International Publishing, 2015.
[8] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering*, 41(9):866–886, 2015.
[9] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In *Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 514–525, 2016.
[10] H. Bagheri and K. Sullivan. Monarch: Model-based development of software architectures. In *Proceedings of the 13th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, pages 376–390, 2010.

[11] H. Bagheri and K. Sullivan. Pol: Specification-driven synthesis of architectural code frameworks for platform-based applications. In *Proceedings of the 11th ACM International Conference on Generative Programming and Component Engineering (GPCE'12)*, pages 93–102, Dresden, Germany, 2012.

[12] H. Bagheri and K. Sullivan. Model-driven synthesis of formally precise, stylized software architectures. *Formal Aspects of Computing*, 28(3):441–467, Mar. 2016.

[13] H. Bagheri, K. Sullivan, and S. Son. Spacemaker: Practical formal synthesis of tradeoff spaces for object-relational mapping. In *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering*, pages 688–693, San Francisco Bay, USA, 2012.

[14] H. Bagheri, C. Tang, and K. Sullivan. Trademaker: Automated dynamic analysis of synthesized tradespaces. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 106–116, New York, NY, USA, 2014. ACM.

[15] A. L. Baroni, C. Calero, M. Piattini, and O. B. E. Abreu. A formal definition for ObjectRelational database metrics. In *Proceedings of the 7th International Conference on Enterprise Information System*, 2005.

[16] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. QAGen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 341–352, New York, NY, USA, 2007. ACM.

[17] N. Bruno, S. Chaudhuri, and D. Thomas. Generating queries with cardinality constraints for DBMS testing. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1721–1725, 2006.

[18] L. Cabibbo and A. Carosi. Managing inheritance hierarchies in Object/Relational mapping tools. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05)*, pages 135–150, 2005.

[19] A. Cunha and H. Pacheco. Mapping between alloy specifications and database implementations. In *Proceedings of the Seventh International Conference on Software Engineering and Formal Methods (SEFM'09)*, pages 285–294, 2009.

[20] B. Demuth, H. Hussmann, and S. Loecher. OCL as a specification language for business rules in database applications. In *Proceedings of the UML 2001–The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 104–117, 2001.

[21] M. L. Drago, C. Ghezzi, and R. Mirandola. A quality driven extension to the QVT-relations transformation language. *Computer Science - Research and Development*, 2011.

[22] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2003.

[23] Ethan K. Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert, and Thomas Santen. Components, platforms and possibilities: Towards generic automation for MDA. In *Proceedings of International Conference on Embedded Software*, 2010.

[24] R. Gabriel. Design beyond human abilities. In *Proc. of the Fifth International Conference on Aspect-Oriented Software Design (AOSD)*, 2006.

[25] S. Gulwani. Dimensions in program synthesis. In *Proc. of PPDP*, 2010.

[26] Harith T. Al-Jumaily, Dolores Cuadra, and Paloma Martínez. OCL2Trigger: deriving active mechanisms for relational databases using model-driven architecture. *Journal of Systems and Software*, 18(12):2299–2314, 2008.

[27] F. Heidenreich, C. Wende, and B. Demuth. A framework for generating query language code from OCL invariants. *Electronic Communications of the EASST*, 9, Nov. 2007.

[28] S. Holder, J. Buchan, and S. G. MacDonell. Towards a metrics suite for Object-Relational mappings. In *Proc. of Model-Based Software and Data Integration (MBSDI'08)*, pages 43–54, 2008.

[29] C. Ireland, D. Bowers, M. Newton, and K. Waugh. Understanding object-relational mapping: A framework based approach. *International Journal on Advances in software*, 2:202–216, 2009.

[30] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[31] W. Keller. Mapping objects to tables - a pattern language. In *Proc. of the European Pattern Languages of Programming Conference*, 1997.

[32] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. Query-aware test generation using a relational constraint solver. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 238–247. IEEE Computer Society, 2008.

[33] S. Krishnamurthi, K. Fisler, D. J. Dougherty, and D. Yoo. Alchemy: transmuting base alloy specifications into implementations. In *Proceedings of FSE'08*, pages 158–169, 2008.

[34] S. Q. Lau. *Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates*. Master's thesis, University of Waterloo, Canada, 2006.

[35] S. Philippi. Model driven generation and testing of object-relational mappings. *Journal of Systems and Software*, 77(2):193–207, 2005.

[36] C. Riva, M. J. Suárez-Cabal, and J. Tuya. Constraint-based test database generation for SQL queries. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 67–74, Cape Town, South Africa, 2010. ACM.

[37] J. Sánchez and G. T. Leavens. Reasoning tradeoffs in languages with enhanced modularity features. In *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, pages 13–24, New York, NY, USA, 2016. ACM.

[38] T. Saxena and G. Karsai. MDE-based approach for generalizing design space exploration. In *Proceedings of the 13th international conference on Model driven engineering languages and systems*, MODELS'10, pages 46–60, 2010.

[39] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL'10*, pages 313–326, Jan. 2010.

[40] S. Srivastava, S. Gulwani, and J. S. Foster. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer*, Jan. 2012.

[41] S. E. Stemler. A comparison of consensus, consistency, and measurement approaches to estimating interrater reliability. *Practical Assessment, Research and Evaluation*, 9(4), 2004.

[42] C. Tang, K. Sullivan, and H. Bagheri. Specification-driven tradespace analysis. Technical report CS-2015-1, University of Virginia, Department of Computer Science, Feb. 2015.

[43] E. Torlak. Scalable test data generation from multidimensional models. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, 2012.

[44] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proc. of Onward*, pages 135–152, 2013.