Towards Designing Effective Data Persistence through Tradeoff Space Analysis

Chong Tang
Department of Computer Science
University of Virginia
Charlottesville, VA 22903.
Email: ct4ew@virginia.edu

Hamid Bagheri
Computer Science and Engineering Dept.
University of Nebraska
Lincoln, NE 68510.
Email: bagheri@unl.edu

Kevin Sullivan
Department of Computer Science
University of Virginia
Charlottesville, VA 22903.
Email: sullivan@virginia.edu

***Abstract*—Abstract specifications span spaces of designs that vary in relevant but under-specified properties. For example, multiple database schemas could store an object model but vary in runtime performance and understandability. In this paper, we present an empirical study to characterize tradeoff decisions made by three popular object-relation mapping tools: Hibernate, Django and Rails. We developed a tool to automate the analysis of these properties. It takes an object model as the input and outputs all schemas and their properties in a design space. We measured understandability by checking how close a schema's structure to its object model. We then find where the schemas produced by each of the ORM tools falls within the design space explored by our analysis. Our study characterizes the schema design spaces for five object models and two popular relational database management systems: MySQL and PostgreSQL. Our results suggest that Rails and Django trade away significant degrees of runtime performance while preserving ease of schema understanding, while Hibernate yields designs on the Pareto front for runtime performance, but with schema structures that are further from isomorphic to the object models they represent, which reduces understandability. No design achieved both near-optimal runtime and understandability.**

I. Introduction

Software development usually starts with collecting user requirements and form a system specification that describe various system properties, like functionality, security, and scalability. Based on the specification, software developers then make a set of decisions about how to translate such specifications to running systems and hope the resulting ones are good enough in satisfying different stakeholders. In practice, however, system specifications are usually partial and incompleteness with respect to design details and the totality of desirable system properties. Such incompleteness creates degrees of flexibility in making decisions in developing software systems, like which algorithms to use and which database system to store application data. Different sets of decisions produce different systems which generally satisfy the given properties but vary in un- and under-specified, often non-functional, properties.

In the domain of object-relational mapping (ORM), an object-oriented data model serves as a specification which constrains the behavior of a persistent data store, it does not uniquely determine the database schema (design); nor does it express preferences for performance properties or tradeoffs involving understandability, read and write performance, and file sizes, which, in general, can vary greatly with the choice of schema.

Developers often rely on methods provided by popular tools to create systems. We call such approaches as *single-point strategy* since they 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. The *single-point strategy* is often supported by design methodologies, tools, or even just training and experience. When tools automatically produce implementations, they often use single-point strategies. Popular ORM frameworks and tools like Hibernate and Rails also provide such strategies. They map object models to relational schemas and code for managing application data. They often use a single mapping strategy, and do not help engineers understand available solutions or the tradeoffs in runtime performance and understandability that they entail.

The alternative approach, which we call *tradespace analysis strategy*, is to consider design spaces, estimate or measure their properties, then choose one that realizes a desirable set of properties. The fundamental *hypothesis* driving the development and use of systematic and automated tradespace analysis is that *it produces results of greater value even net of its additional costs*. Indeed, tradespace analysis tends to reveal designs, that otherwise engineers might miss, with properties and tradeoffs that are significantly better than those that engineers would otherwise obtain (e.g., using single-point methods).

In this paper, we present the results of an empirical study of evaluating the runtime performance and understandability of database schemas created by three popular ORM tools: Hibernate, Rails, and Django. We studied three runtime performance: data population, data retrieval, and disk space consumption, and understandability of the created schemas.

We approximate the understandability of a schema with three criteria suggested by Holder et al. [1] and Baroni et al. [2]: 1) Number of Corresponding Relational Fields (NCRF); 2) Number of Involved Classes (NIC); and 3) Referential Integrity Metric (RIM).

The approach we used is a *tradespace analysis strategy*. Given an object model, we first create a formal representation in a domain specific language (DSL). Then, we leverage software synthesis techniques to generate the whole tradeoff space consists with all possible schemas and corresponding test cases. Next, we calculate the understandability with the criteria discussed above and dynamically evaluate the runtime performance of all created schemas using the MapReduce framework. At last, we plot where the *single-point schemas* created by the three ORM tools fall in the tradeoff space and thus study their tradeoff among the evaluated properties.

We studied the schema design spaces for five object models and two popular relational database management systems: MySQL and PostgreSQL. Our results suggest that Rails and Django trade away significant degrees of runtime performance while preserving ease of schema understanding, while Hibernate yields designs on the Pareto front for runtime performance, but with schema structures that are further from isomorphic to the object models they represent, which reduces understandability. There is no design achieved both near-optimal runtime performance and understandability.

To support such tradespace evaluation, we developed a novel framework that leverages recent advancements in several areas, such as constructive logic, formal synthesis and scalable big data analytics. More specifically, this paper contributes a formal, general algebraic theory of design space tradeoff analysis tools, and a MapReduce-based framework, derived mechanically from the theory, for implementing such tools. The theory is organized as a hierarchy of Coq typeclasses. From this theory, we automatically derive a polymorphic framework (in Scala) that developers specialize and extend to produce domain-specific trade-off analysis tools.

To summarize, this paper makes the following contributions:

- *Theoretical framework:* We develop a theoretical framework conceptualized in constructive logic to make the notion of tradeoff analysis precise and practical.
- *Empirical study:* We show that none of the studied ORM framework can make good tradeoffs among properties valued by stakeholders with a set of experimental results.

The rest of this paper is organized as follows. Section 2 motivates our research through an illustrative example. Section 3 provides an overview of ASTRONAUT. Section 4, 5 and 6 describe the details of our constructive logic based framework of tradeoff analysis, framework instantiation, and parallelization reasoning, respectively. Section 7 presents the evaluation of the research. Finally, the paper concludes with an outline of the related research and our future work.

II. MOTIVATION AND RUNNING EXAMPLE

In this section, we discuss how incompleteness in specification gives rise to important separations of concerns and the need for a systematic understanding and application of tradeoff analysis. We then present a running example having to do with tradeoff analysis among the space of possible solution alternatives.

A. Strategic Incompleteness in Specification

Specifications are often incomplete with respect to the full range of properties that stakeholders value in a given system. Such incompleteness is often not a flaw. Rather, it can serve a strategic function in structuring the process of complex system design. When a specification is silent on system properties relevant to stakeholders, it partitions the design process, the representation of acceptable solutions, and the set of design decisions to be made. We address each of these separations of concerns and explain how they create a need for a better theory of and technology for tradeoff analysis.

Partitioning of the Design Process. Incompleteness partitions the design process into at least two distinct parts. The first is a deductive process, in which candidate solutions are derived from a specification, constrained only by the condition that they satisfy its terms. Such solutions generally differ in stakeholder-relevant properties on which the specification was silent. The second part is thus an optimization process, in which solutions are evaluated for additional properties, tradeoffs are identified, candidates are ranked, and one is selected for use or development.

Partitioning of Design Representations. This deductive vs. optimization partitioning of the design process is mirrored by an explicit vs. implicit partitioning of the representation of what constitutes an acceptable solution. The explicit part is given by the specification. The implicit part is represented in the property estimation functions that will be used to evaluate solutions, the stakeholder utility functions (emergent or documented) that map property estimates to stakeholder utilities, and the stakeholder tradeoff functions (emergent or documented) that map the multiple stakeholder utilities to a final ranking of, and ultimately to a choice from among, candidate design solutions.

Partitioning of Design Decision Spaces. The deductive vs. optimization and explicit vs. implicit dichotomies extend to a split between decisions that are understood and agreed on well enough to be pinned in a specification, and those that are not. This is a split between settled vs. unsettled decisions. A specification speaks explicitly on design decisions that are settled while remaining silent on relevant but as yet unsettled aspects, leaving them to be worked out in downstream, optimization-oriented design activities.

These separations of concerns can also sometimes be seen in the evolution dynamics of complex systems. As initially

unsettled concerns are settled, they can migrate from being represented implicitly in measurement and utility functions to being explicit in specifications. System architectures can be seen as settled and explicit specifications, for example, that remain incomplete in other key areas. As optimization-based processes produce knowledge and agreement, these results can migrate into specifications.

B. Running Example

To further motivate the research and illustrate our approach, here we provide a running example having to do with tradeoff analysis among the space of possible solution alternatives. Consider object-relational database mapping. In ORM domain, a specification is an object model that contains objects and relationships among these objects. Object-oriented data models serve as specifications for application database schemas. While these specifications constrain schemas, they are silent on properties such as time and space performance, and evolvability. At the same time, degrees of freedom in ORM mappings (e.g., in how inheritance is mapped to relations) give rise to spaces of satisfying schemas that vary in these properties. Object-oriented specifications of data models are incomplete regarding these other properties. While all solution points within the solution space are equally good at satisfying the specification, they will differ in other properties of interest in design.

Incompleteness is generally resolved today by policies hard-wired into ORM packages. One straightforward solution is created for any given specification, without much consideration of stakeholder preferences. Such tools impose tradeoffs on stakeholders that might or might not be desirable.

The above example points to one of the most prominent and widely-used software/system engineering problems that we take as a running example that requires tradeoff analysis to be addressed effectively. The upshot is that tradespace analysis is an important part of practical design, in general. The motivation for this paper is the current lack of adequate scientific foundations and technologies for tradespace analysis in software and systems engineering. The consequences are significant, in opportunity costs, stakeholder dissatisfaction, and in underperforming and failed projects and systems.

III. Approach Overview

This section presents an overview of our approach for automated tradeoff analysis, and describes how it leverages advances in several areas of computer science and engineering.

A. Constructive Logic & Certified Programming with Dependent Types

First, we use the enormous expressiveness of dependent type theory in modern constructive logic proof assistants to produce precise, yet computationally effective, theoretical framework for tradeoff analysis. We present an algebraic theory of tradeoff analysis tools structured as a hierarchy of Coq [3] typeclasses, in a style similar to that being used by mathematicians [4], [5] to formalize hierarchies of abstract algebraic structures (e.g., groups, fields, topological spaces).

From this theoretical framework, we use Coq's extraction facility to derive a certified [6] implementation of a general-purpose framework in Scala. It is then specialized and extended with user-defined, domain-specific types and functions, subject to the specified laws, to create domain-specific analysis tools. The framework provides implementations of functions common to all instances and expresses laws that all instances must obey.

B. Formal Synthesis from Specifications

Our framework is meant to be specialized using any types of software synthesis techniques. In this work, we specialize it in the context of database schema designs for object-oriented applications. We use a relational logic model finder to exhaustively synthesize relational database schemas as well as test loads for dynamic analysis of performance from specifications of object-oriented data models [7], [8]. With this tool, we are now able to fully replicate the largely manual analysis of synthesized database schemas reported in our earlier work [8].

C. Scalable Big Data Analytics

We use big data analytics, particularly map-reduce [9], to reduce analysis runtimes. While synthesizing spaces of design solutions from specifications may not always be easily parallelized, applying property analysis functions to dynamically measure each design solution in multiple dimensions of performance is. In fact, applying each measurement function to each solution has a natural map-reduce structure. The use of scalable map-reduce technology can benefit many instances of our framework, so it is practical to support it as a common middleware plug-in. In the following sections, we describe the details of our approach.

IV. A Constructive Logic Based Framework of Tradeoffs

This section presents our framework of tradeoff analysis tools as a hierarchy of typeclasses in Coq. We identified that such a framework must support: 1) different types of formal specification inputs, which we take to be incomplete in general regarding all properties of interest; 2) different kinds of synthesizers to solve the given specification to produce implementations; 3) different kinds of test loads for dynamic analysis, and specialization of common loads to the interfaces exposed by particular implementations; 4) different types of system property measurement functions; 5) different types of analysis results; and 6) different kinds of filters to select optimal designs based on analysis results.

We have exploited the idea of using typeclasses to capture general families of mathematical structures by having our typeclass-based specification define the *family* of Astronaut

instances, including dimensions of variation, shared operations, and invariants common to all family members. Typeclasses in Coq are useful for specifying interrelated *families* of mathematical structures, including carrier sets, operations, and invariants, as well as for specifying instances of these structures. An example of interrelated families of structure from abstract algebra include monoids, groups, and fields (each of which enriches the previous one).

The structure of our framework is shown in more detail in Figure 1. It takes user defined specifications as inputs. The synthesizer then synthesizes a set of implementation and associated test loads. Next, the measurement function dynamically analyzes each implementation with associated test loads to get a set of measurement results for each solution point. At last, the filter picks the set of (Pareto-)optimal solutions based on the measurement results.

A. Astronaut Typeclass

Astronaut is the most abstract, least structured typeclass in our hierarchy. It expresses the functionality of a broad family of tradespace analysis tools at a high level of abstraction. The code fragments in Listing 1 presents the hierarchy of abstract algebraic structures that we formalized as Coq typeclasses. Each typeclass (Set in the code) characterizes a class of possible instances. Each class represents a type, values of the type, operations on the type, and laws constraining the behaviors of the operations. It has four types: (1) type of input specification (e.g., a UML class diagram); (2) type of derived implementation (e.g., SQL schema); (3) type of a set of property measurement functions (e.g., space performance measurement); and (4) type of measurement results (e.g., database size in megabytes). Values have to be provided when instantiating Astronaut typeclass.

The *synthesize* component is a function that maps a specification to lists of $< Implementation, MeasurementFunction >$ pairs. The measurement functions provide performance comparisons of variant implementations. The *mapReduce* component is a function that takes a list of $< Implementation, MeasurementFunction >$ pairs, and returns a list of $< Implementation, MeasurementResult >$ pairs. The *astronaut* function calls the *map* function to map the measurement functions over the output of the synthesize function (a list of $< Implementation, MeasurementFucntion >$ pairs). Implementations of these functions are required when the typeclass is instantiated.

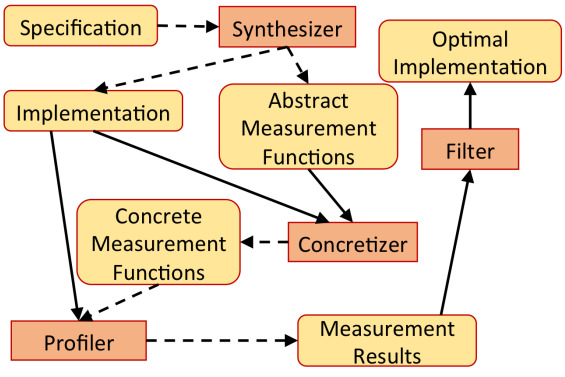

Fig. 1: Framework Data Flow

```
1  Class Astronaut := {
2    SpecType: Set
3  ; ImplType: Set
4  ; MeasureFuncSetType: Set
5  ; MeasureResultSetType: Set
6  ; synthesize : SpecType ->
7      list(ImplType × MeasurFuncSetType)
8  ; mapReduce: list (ImplType ×
9          MeasureFunctSetType) ->
10     list (ImplType × MeasureResultSetType)
11 ; astronaut (spec: SpecType):
12     list (ImplType × MeasureResultSetType) :=
13     map mapReduce (synthesize spec)
14 }.
```

Listing 1: Astronaut Typeclass Coq specification, which defines the most abstracted framework structure. It takes a specification as input, and outputs a set of implementations and measurement results.

B. Tradespace Typeclass

The Astronaut typeclass[1] (Listing 1) captures a very general notion of tradespace analysis, and nicely illustrates some of the aspects of our approach, but it provides too few details for implementing a tradespace analysis tool. We introduce a new tradespace typeclass to enrich the Astronaut typeclass to provide a finer-grained implementation architecture for tradespace analyzers.

Listing 2 partially presents Tradespace typeclass Coq specification. The first two lines state that the *Tradespace* class extends (is coercible to) the Astronaut and ParetoFront typeclasses. The latter provides structure for computing Pareto fronts of sets of vector-valued objects. The following four components (lines 5–8) provide for parameterization of typeclass instances with respect to the key additional types of the implementation framework. Following the declarations of these type-valued parameters, lines 10–21 specify the signatures of the mapping functions required to instantiate the Tradespace typeclass.

[1]Research artifacts and the complete model for ASTRONAUT, including all specifications, are available at http://chongtang.github.io/Astronaut/

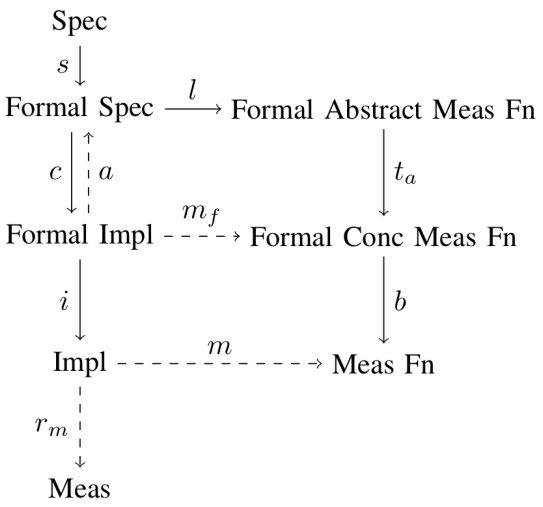

Fig. 2: Tradespace typeclass model

```
1  Class Tradespace := {
2    tm_Astronaut :> Astronaut
3  ; tm_ParetoFront :> ParetoFront
4  (* Internal, Formal-Spec-based types *)
5  ; FmSpecType: Set
6  ; FmImplType: Set
7  ; FmAbsMeasureFuncSetType: Set
8  ; FmConcMeasureFuncSetType: Set
9  (* Internal, Formal-Spec-based functions *)
10 ; cFunction: FmSpecType -> list FmImplType
11 ; aFunction: FmImplType -> FmSpecType
12 ; lFunction: FmSpecType ->
13                FmAbsMeasureFuncSetType
14 ; tFunction: FmAbsMeasureFuncSetType ->
15                list ImplType ->
16                list FmConcMeasureFuncSetType
17 (* map to and from Formal-Spec-based form *)
18 ; sFunction: SpecType -> FmSpecType
19 ; iFunction: FmImplType -> ImplType
20 ; bFunction: FmConcMeasureFuncSetType ->
21                MeasureFuncSetType
22 (* Laws *)
23 ; aInvertsC: forall (spec: FmSpecType)
24                (fImpl: FmImplType),
25                In fImpl (cFunction spec) ->
26                (spec = aFunction fImpl)
27 (* See code repo, omission on purpose. *)
28 ; implLineLaw: ...
29 (* See code repo, omission on purpose. *)
30 ; testLoadsLineLaw: ...
31 }.
```

Listing 2: Tradespace Typeclass Coq specification. It captures the internal structure of our framework.

The diagram shown in Figure 2 graphically depicts the structure of the Tradespace typeclass. The concretization function (cFunction) maps the formal specification to a set of formal representations of implementations (FmImplType). The function abstraction (aFunction) explains how each implementation represents and satisfies its specification. The load function (lFunction) maps the same formal specification to a set of abstract measurement functions (FmAbsMeasureFuncSetType) that will be used to produce concrete measurement function (FmConcMeasureFuncSetType) to measure the properties of implementations. The function tFunction is a conceptual function serves our explanation here and doesn't require implementation. It specializes the abstract measurement functions to the particular interfaces exposed by variant implementations. This function uses the abstraction function aFunction to do its work. The result of this process is another conceptual function/relation, as indicated by m_f in Figure 2, that associates a vector of implementation specific measurement function(s) to each implementation.

The function sFunction translates a user given specification to an internal formal specification that can be solved by cFunction. The function iFunction parses the formal/internal representation of implementations to usable forms: e.g., Alloy solutions to SQL schemas. The function bFunction similarly parses formal/internal representation of measurement functions to useful forms: e.g., to objects that run actual SQL scripts against actual databases. The final tradespace analysis result is the relation r_m in Figure 2 that associates implementations with their corresponding property measurement vectors.

The final three components (lines 23–30) specify laws that the other components of the typeclass must follow. In a nutshell, these laws state that the abstraction function, a, must invert the concretization function, c, and that the two paths from specification to measurements must yield the same results. Constructing an instance of the typeclass requires proofs of these propositions for the given function and data type parameter values.

V. Framework Instantiation

Framework users need to provide domain-specific types for the nodes in Figure 2 and domain-specific function implementations for the solid arcs. The other dashed-line mappings are implicit or automatically computed. This section describes how we instantiate our framework to create an automation tool in the context of the ORM domain.

To instantiate an automated tool based on the framework for ORM tradespace analysis, we defined a $DBFormalSpec$ class as an actual parameter for the $FormalSpec$ slot in this architecture (cf. Fig. 2). Concretely, it is a wrapper around a file containing Alloy specification of an object model. Similarly, we created a $DBImplementation$ class that wraps a file containing a MySQL schema definition, as a parameter for the $Impl$ in Figure 2. Our implementation of the function c realizes our Alloy-based approach to synthesize database schemas from object models. Our other ORM-specific values

are similar in their structure: classes (Scala types) wrap representation details and function implementations such that they encapsulate details of computations of the various mappings required to implement our tradespace analysis approach. We also implemented $mapReduce$ function in Astronaut typeclass using Spark library for Scala. When the tool actually runs, it sends measurement jobs to a pre-configured Spark cluster.

We then provided several parameter values to instantiate the framework. The components include: 1) An Alloy-based ORM domain specification language; 2) an Alloy-based synthesizer to generate candidate designs and test loads (INSERT and SELECT SQL scripts) to be executed to dynamically analyze database designs; 3) two kinds of measurement functions (*Time* and *Space* measurement functions) to measure time and space tradeoffs; 4) a triple of population time, retrieval time, and space consumption as a group of analysis results; and 5) a Pareto-optimal filter to filter out non-Pareto-optimal designs.

Our instance for ORM analysis of this framework is produced with the following parameters.

- SpecType: Object models in formal Alloy-based notation
- ImplType: SQL schema
- MeasureFuncSetType: an instrumented test harness for profiled execution of synthesized SQL scripts
- MeasureResultSetType: a tuple of time and space performance measures from instrumented benchmark execution
- synthesize: given an object model, produce a list of SQL schema and SQL script pairs (INSERT and SELECT statements)
- mapReduce: run a profiled SQL script on a database with the given schema in map-reduce style
- sFunction: An identity function (since the Specification Type in our case is already a formal object model)
- cFunction: Alloy based analyzer that synthesizes an object model to a set of database designs in the form of database creation scripts
- aFunction: A function that returns the mapping information in each database design
- lFunction: A generator that creates formal abstract measurement functions from an object model
- tFunction: A concrete formal measurement function specializer that mapping a formal abstract measurement function to a set of concrete measurement functions based on mapping information of each designs
- iFunction: A parser that parses the database schema embedded in Alloy solution (XML files) to SQL database initial script
- bFunction: An identity function (the formal concrete measurement function in our case is already INSERT and SELECT SQL statements)

Algorithm 1: Profiling a given design solution in a worker node.

Input: i: Formal Impl, **// design to be analyzed**
f_a: AbstractionFn, **// abstraction function**
M_a: List<AbstractMeasFn>
// abstract measurement functions
Output: p: Result **// profiling evaluation results**

1: $M_c = newList()$ **// concrete measurement functions**
2: **for** $m_a \in M_a$ **do**
3: $m_c \leftarrow generateConcreteMeasure(m_a, f_a, f)$
4: $M_c.add(m_c)$
5: **end for**
6: $profResults \leftarrow newList()$
7: **for** m_c in M_c **do**
8: **for** i in $[1:3]$ **do**
9: $result \leftarrow runMeasureFunction(i, m_c)$
10: $profResults.add(result)$
11: **end for**
12: **end for**
13: $p \leftarrow average(profResults)$
14: **return** p

VI. Parallelization Reasoning

Even simple specifications could map into a vast number of solutions in the design space. The combination of ample computing capacity and our ability to synthesize immense solution spaces for given specifications suggests that we measure properties and tradeoffs of actual running systems in the spirit of what Cadar et al. called multiplicity computing [10], with the goal of producing new speed-ups in tradeoff analysis of real-world software systems.

Recall that tradeoff analysis using our approach consists of three steps: (1) Solution space and abstract measurement functions are synthesized from formal specifications. (2) The synthesized implementations and abstract measurement functions are transformed into concrete formats, e.g., here formal models are concretized into database schemas and corresponding test loads. (3) Synthesized solution spaces are dynamically analyzed by profiling performance of each solution under its corresponding structure-specific measurement function. In this section, we focus on steps 2 and 3 that can significantly benefit from parallelizing the process.

Similar to a conventional cluster computing paradigm, Astronaut's approach for parallelization consists of a large number of worker machines managed by a master node referred to as the cluster manager. Once Astronaut's cluster manager deploys a synthesized solution to a worker machine, it runs to profile the performance of the given design solution in parallel with other worker machines.

Algorithm 1 outlines the process of profiling a given design solution as realized in a worker node. It takes as input

(1) a design solution to be analyzed, i, (2) its associated abstraction function, f_a, that explains how the given design solution instantiates its abstract specification, and (3) a vector of measurement functions, which are essentially abstract test loads automatically generated from the specification and need to be concretized for the given design solution.

The logic of the algorithm is as follows. For each abstract measurement function, $m_a \in M_a$, generate concrete measurement functions, or test loads, that meet the particular structure of the given design solution, i. In doing so, $generateConcreteMeasure$ relies on the abstraction function that relates the design solution back to the abstract specification from which input measurement function, or abstract test load, is derived, and that from these abstraction functions it derives functions for concretizing the given abstract loads. After the measurement functions are generated for the given design, run them to profile its performance.

VII. Experimental Evaluation

To assess the effectiveness of our theory and framework in supporting tradeoff analysis, we have conducted an experimental evaluation that addresses the following research questions:

- **RQ1.** Does ASTRONAUT tradespace analysis enable production of Pareto-optimal designs that are entirely overlooked by widely-used, industrial ORM tools?
- **RQ2.** How well does ASTRONAUT perform? What is the performance improvement achieved by ASTRONAUT's designs compared to those produced by industrial frameworks?
- **RQ3.** What is the overhead of ASTRONAUT in conducting tradespace analysis?

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

A. Experimental Objects

Our experimental subjects are 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 first is the object model of an E-commerce system adopted from Lau and Czarnecki [11]. 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 the CSOS system, a 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 object model is the object model of a documents sharing application called Flagship Docs built with Ruby on Rails at Rensselaer Polytech [12]. It has 6 classes connected by 8 associations with no inheritance relationships. We also analyzed an extended version of our customer-order example [8].

The selected experimental subjects are representatives of large classes of useful applications at a scale matched to the state-of-the-art synthesis techniques. Their databases have multiple tables, and several relationships among these tables. There are multiple ownership relationships among these tables, which induce many possible choices of object-model to relational schema mappings.

B. Experimental Setup and Data Collection

We followed several steps to address the research questions. First, we wrote database specifications using Django, Rails, and the object-modelling language developed by Bagheri et al. [8], for each one of the five systems. We chose to focus on time and space performance of *CREATE* and *READ* transactions for dynamic analysis of target database performance. We wrote instrumentation code to return the desired time and space consumption data into the map-reduce computation. We used the *mysql* client command to execute synthesized test loads, which our tool parses from internal Alloy representations into MySQL scripts. The first sets of statements were database structure creation scripts. Then *insert* scripts were executed to populate each database with the test data generated in previous stages. The experimental data collected includes the time to run these commands as well as the space consumed by the databases. After populating the databases with test data, we executed *READ* transactions, which are essentially collections of *select* commands. We ran each script three times to rule out other uncontrolled factors that might influence the data we collected. The evaluation results are a list of triples: insertion time, retrieval time, and space consumption. The final result reported in this paper is the average time and space performance.

C. Astronaut vs. Industrial Platforms

Figure 3 depicts tradeoff analysis plots produced by ASTRONAUT. It projects the results into 2-D tradeoff plots: Each plot presents tradeoff information in two of the three dimensions for one subject system. Each dot in the plots represents the analysis result of one schema. The hollow triangles that connected by a line are the Pareto-optimal designs in corresponding tradeoff space, exhibiting Pareto-optimal solutions in each plot. We also find out the two schemas produced by *Rails* and *Django* from the synthesized schemas, by looking up the structure in SQL scripts. We then locate the analysis results of these two schemas, and mark them in the 2-D plots with different color and shapes. The hollow star is the design created by Django. The hollow diamond depicts the design generated by Rails. One can easily

distinguish the schemas from the other synthesized schemas, indicating where the *Rails* and *Django* generated schemas are fall in the tradespace in terms of the triple measures.

Consider the CSOS Insert-Space tradeoff plot (Row 2 and Column 2) as an example. It shows the tradeoff information of insertion time (in seconds) and space consumption (in megabytes). The two triangle depicts the Pareto front in these two dimensions. It could be just one triangle if there were only one design on the frontier. We can observe that although the schemas created by these tools are not same, their performances outcomes are close together. The most interesting point is that both of the generated schemas by these tools are far from the Pareto fronts. The same phenomena appear in other plots as well.

The experimental data thus suggests that ASTRONAUT tradespace analysis enables production of Pareto-optimal designs that are entirely overlooked by widely-used, industrial ORM tools.

D. Improvements in Practice

Table I presents the performance improvement in the insertion time by comparing a Pareto-optimal schema produced by ASTRONAUT to the performance of databases created by Rails and Django ORM systems. The first line are the five experimental subjects. The second line is the insertion time consumed by the ASTRONAUT's designs. The third line is insertion time consumed by the *Rails* designs, and the forth line is insertion time consumed by *Django* designs. Finally, the last line represents the average performance improvement over *Rails* and *Django* designs across subject systems. Table II and table III similarly tabularize the performance improvement in the retrieval time and space consumption.

From the experimental results, we can observe that the Pareto-optimal designs revealed by our analysis generally have much better performance in all dimensions: insertion times, retrieval times, and space consumption. Based on these analysis results, we conclude that our synthesis technique tends to perform better than commonly used tools, particularly, *Rails* and *Django*. Our synthesized designs have better performance, in three models: *CustomerOrder*, *CSOS*, *ECommerce*, and *Decider*, in all three dimensions. For the *Flagship Docs* model, *Rails* and *Django* produced designs equal to ours in performance. We have ascertained that the reason is that the *Flagship Docs* data model has no inheritance relationship, so all of the objects in the object model are mapped as an independent tables in synthesized database schemas.

E. Performance and Timing

The final evaluation criteria are the performance benchmarks of tradespace analysis. To carry out the experiments, we set up a 16-node Spark cluster as a back-end analysis platform. Each node has a 2-core AMD Opteron(tm) 242 CPU with kernel clock frequency of 1.5GHz, and 3 gigabytes of memory in total, where 1.9 gigabytes of memory is allocated to Spark. The worker nodes have MySQL server and client tools installed.

Subject	CO	CSOS	EComm	Decider	Flagship
ASTRONAUT	60.01	78.89	112.64	82.24	13.23
Rails	97.99	133.61	189.93	108.62	13.23
Django	97.99	135.41	189.93	108.62	13.23
Average Improvement	63.3%	70.5%	68.6%	32.1%	0%

TABLE I: Insertion Performance (Seconds)

Subject	CO	CSOS	EComm	Decider	Flagship
ASTRONAUT	82.12	129.10	131.09	94.46	14.03
Rails	125.54	222.25	231.66	131.57	14.03
Django	125.54	221.25	231.66	131.57	14.03
Average Improvement	52.8%	71.8%	76.7%	39.3%	0%

TABLE II: Retrieval Performance (Seconds)

Table IV shows the analysis time taken to produce tradespaces across subject systems. The first line are the five experimental subjects. The second line represents the size of tradespace for each subject system. Finally, the last line shows the time required to automatically conduct the tradeoff analysis.

The experimental results confirm that ASTRONAUT approach is effective in systematic dynamic tradespace analysis. In this particular case, the generated Pareto-optimal solutions provide crucial performance improvements to both systems designers and their end-users. Such analysis is not possible with state-of-the-practice tools (e.g., Django and Rails) that produce a single point solution. The results are fascinating, especially when compared with the state-of-the-art technique that requires more than two person months of efforts, during which they were able to analyze only a small sample of representative designs [8]: 14 for CustomerOrder, 121 for CSOS, 154 designs for the Decider, and 360 for the ECommerce model. In fact, ASTRONAUT automates the complete dynamic analysis of exhaustively enumerated design spaces. Clearly, Astronaut does vastly reduce the time required for exhaustive dynamic tradeoff analysis. We expect that performance of ASTRONAUT improves by leveraging more capable computing resources, such as Amazon Web Services, for its parallel reasoning.

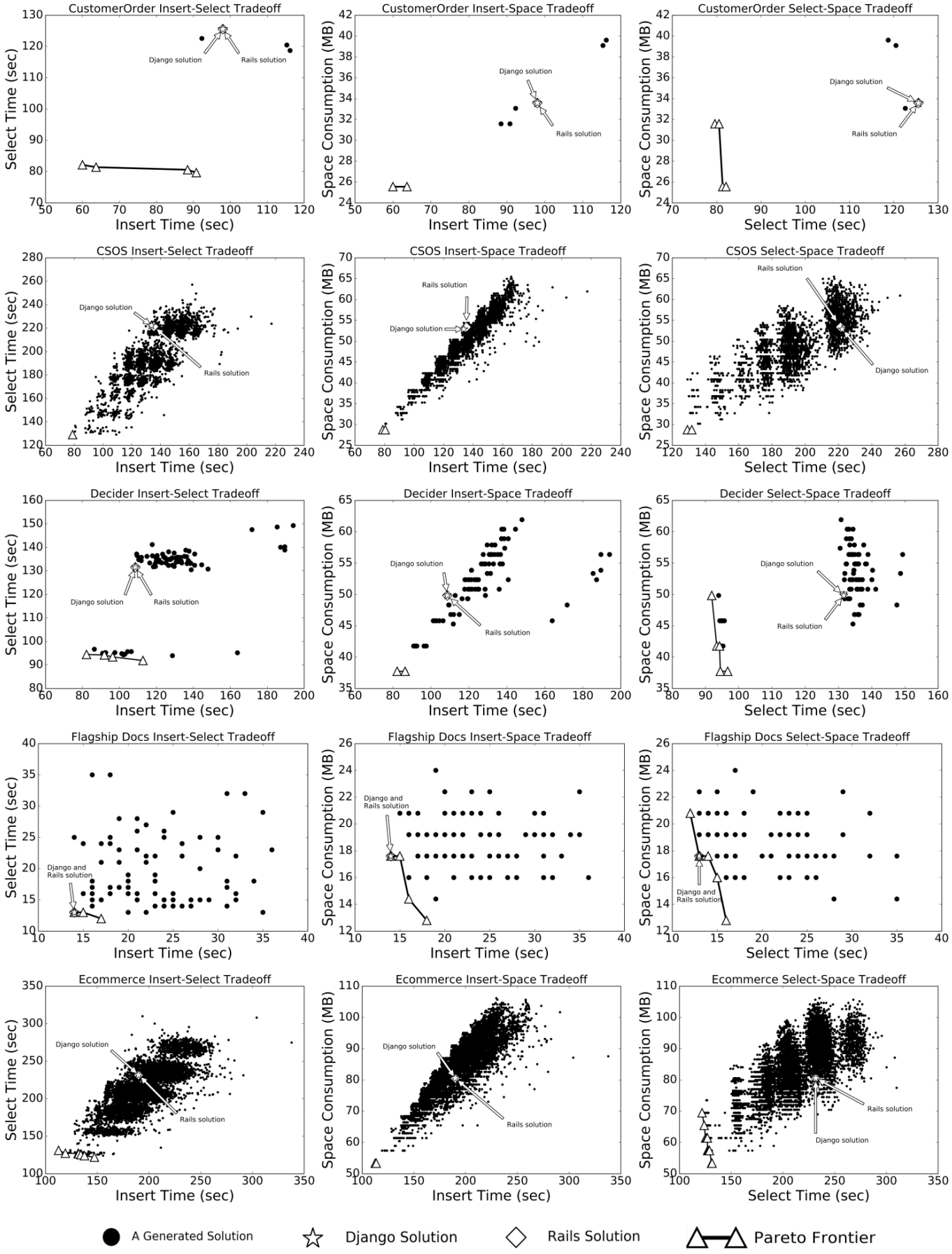

Fig. 3: Tradeoff analysis results; columns represent tradeoff diagrams across systems in two dimensions of Insert-Select, Insert-Space and Select-Space from left to right, respectively; each black dot represents performance of a synthesized database schema, and the star and diamond entries plot where the results of schema generated by Django and Ruby fall, respectively; both star and diamond entries are far from the Pareto fronts generated by ASTRONAUT.

Subject	CO	CSOS	EComm	Decider	Flagship
ASTRONAUT	25.55	28.72	53.34	37.73	17.64
Rails	33.58	52.86	80.56	49.83	17.64
Django	33.58	53.36	80.56	49.83	17.64
Average Improvement	31.4%	85.0%	51%	32.1%	0%

TABLE III: Space Consumption (MB)

Subject	CO	CSOS	Decider	Ecomm	Flagship
Solutions	28	3872	144	14400	256
Time	4 m	5.2 h	15 m	17.8 h	17 m

TABLE IV: Analysis time to produce tradespaces across subject systems

VIII. DISCUSSION

The experimental results make it clear that techniques relying on the single-point strategy, such as conventional ORM tools, produce solutions with strictly and significantly sub-optimal performance relative to actual Pareto fronts, and that ASTRONAUT is capable of producing strictly and significantly superior designs.

Our framework makes the shared architectural and computational structure of a family of similar tools explicit, including its map-reduce computational structure and the filtering of results that are strictly dominated by other results. We captured the general structure of this whole tool family using Coq typeclass. Parameters of Coq typeclasses allow variation in the stated dimensions, with propositional laws capturing (some of) the required semantics of the components that one "plugs in" to our framework.

There is a growing need for technologies that can support systematic tradeoff studies to reveal, among others, how system properties in multiple dimensions vary across implementations, how stakeholders might be impacted, and what implementations might best serve the needs of a given project. We argue this is the holy grail of software design research. Astronaut takes an important step towards this overarching objective by helping engineers understand important tradeoffs among dynamically measurable properties for important classes of software designs, at meaningful scales within reach of existing synthesis technologies. We envision the ideas set forth in this research to find a broader application in other computing domains as well.

A comment on scalability is in order. Relational-logic tradeoff analysis involves exhaustive enumeration of models of bounded relational logic models. This is a *#P-complete* problem: harder than *NP-complete* and equivalent to counting the number of satisfying solutions to a SAT problem. It is intractable in general, and will not scale to complex, integral systems. Yet, model checking tools have clearly demonstrated the potential value in exploring practical uses of solutions to theoretically intractable problems. Relational-logic tradeoff analysis is no panacea. In this work, we observed its potential utility for problems at the scale of individual modules, such as database schemas for ordinary web applications. While the modular architecture of ASTRONAUT supports variation in logics and solvers thus enumeration strategies, we leave the exploration of such variations on our theme to future work.

IX. RELATED WORK

We can identify in the literature three categories of research that are related to ours: software synthesis, tradeoff analysis, and search-based software engineering.

Software synthesis. There is a large body of research on synthesis techniques. Dang [13] provided a tool for embedded software synthesis. Andersen [14] provided a framework for mathematical problems synthesis. Le [15] provided a framework for data extraction from different kind of sources, like text file, webpages, and spreadsheets. Gupta [16] provided a high-level synthesis framework to transfer a behavioral description in ANSI-C to register-transfer level VHDL with parallel compiler transformation technique. Assayed [17] provided a synthesizer for multi-threads software on multi-processor architectures. Our work in this paper is aimed to provide a general framework for both tradeoff synthesis and analysis. The pluginable feature enables the support of arbitrary DSLs and different forms of final implementations. The users can create their own instance by just providing necessary components.

The work in [18] synthesizes program according to the pre- and post-conditions of program functions. Sketeching [19] similarly 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 incomplete specifications and synthesis based on constraint solving. However, it does not perform a tradespace analysis over possible completions of a sketch. The synthesis refers to the concrete example of correct or incorrect behavior to prune the design space, and finally narrow down the design space to one implementation that satisfying the given specification.

Srivastava et al. [20] 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 all these techniques, ASTRONAUT tackles the automated tradeoff space analysis through synthesizing spaces of design alternatives and common measurement functions over such spaces. It thus relieves the tedium and errors associated with their manual development. To the best of our knowledge, ASTRONAUT is the first formally precise and general-purpose technique for automated synthesis and dynamic analysis of tradeoff spaces.

Tradeoff analysis. The other relevant line of research focuses on tradeoff analysis. Petke et al. [21] used Genetic Improvement techniques to transplant code from one version of a system to another, and profiling the new system to enhance execution performance. Schulte et al. [22] optimized software non-functional properties that left behind by compilers. Our work seeks the tradeoff of non-functional properties by both synthesis and analysis of design spaces from a given specification.

Bondarev et al. [23] proposed a framework, called *DeepCompass*, that analyzes architectural alternatives in dimensions of performance and cost to find Pareto-optimal candidates. Their approach, however, requires a manual specification of architectural alternatives, and provides no support for architecture synthesis.

Aleti et al. [24] developed *ArcheOpterix* for optimizing an embedded system's architecture. They applied an evolutionary algorithm to optimize architectures modeled in the AADL language with special focus on component deployment problems. Martens et al. [25] also developed *PerOpteryx* to automatically improve an initial software architectural model through searching for Pareto-optimal solution candidates. They applied a genetic algorithm to the Palladio Component Models of given software architectures.

Like many other research work we studied, these research efforts do not support automatic production of the design tradespaces from incomplete specifications, rather they start from a solution point and tend to gradually optimize it. We see these two types of approaches for tradeoff analysis complementary.

Along the same line, Trademaker [8] studies whether relational logic model finders, such as Alloy, can be used for design synthesis and tradeoff analysis. ASTRONAUT, however, formalizes the notion of tradeoff analysis and contributes a theoretical framework conceptualized in constructive logic, from which a polymorphic implementation framework for tradeoff analysis tools is mechanically derived.

Search-based software engineering. Harman [26] reviewed the application of search and optimize technique in eight software engineering domains, as well as the open problems and challenges. Weimer et al. [27] used search techniques to help find candidate code snippets to repair buggy code. Jia and Harman [28] used automated search-based techniques to find rare but valuable test cases. McMinn [29] surveyed the application of search techniques for test data automatic generation. Our work uses search related techniques to find Pareto-optimal solutions based on the analysis result, to help design better systems.

X. CONCLUDING REMARKS

The state-of-the-art in software engineering methods and tools does not provide adequate support for tradeoff analysis in design spaces created by incomplete specifications. We hypothesized that engineers tend to deal with this situation by settling on designs deemed good enough, but doing so can fail to reveal better designs, thereby leaving significant value untapped.

We have presented ASTRONAUT, the first scientific foundations and a novel, general-purpose software technology for practical tradespace analysis. The key contributions of our work are (1) a theoretical framework conceptualized in constructive logic to make the notion of tradeoff analysis precise, (2) a mechanically derived, polymorphic implementation framework for tradeoff analysis tools, (3) results from experiments in the particular domain of object-relational database mapping, corroborating that widely-used industrial ORM tools appear to produce solutions with performance properties that are far from those achievable through more systematic design space modeling and analysis, and that systematic dynamic tradespace analysis can find much better designs within plausibly practical time frames.

REFERENCES

[1] S. Holder, J. Buchan, and S. G. MacDonell, "Towards a metrics suite for Object-Relational mappings," *Model-Based Software and Data Integration*, vol. CCIC 8, 2008.

[2] 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. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.65.688

[3] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: CoqâĂŹArt: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[4] B. Spitters and E. Van der Weegen, "Type classes for mathematics in type theory," *Mathematical Structures in Computer Science*, vol. 21, no. 04, 2011.

[5] Á. Pelayo, M. A. Warren, and T. V. Voevodsky, "Homotopy type theory," *Gazette des Mathematiciens*, no. 142, 2014.

[6] A. Chlipala, *Certified programming with dependent types: a pragmatic introduction to the coq proof assistant*. MIT Press, 2013.

[7] H. Bagheri, K. J. Sullivan, and S. H. Son, "Spacemaker: Practical formal synthesis of tradeoff spaces for object-relational mapping." in *SEKE*, 2012.

[8] H. Bagheri, C. Tang, and K. Sullivan, "Trademaker: Automated dynamic analysis of synthesized tradespaces," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014.

[9] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, 2008.

[10] C. Cadar, P. Pietzuch, and A. L. Wolf, "Multiplicity computing: a vision of software engineering for next-generation computing platform applications," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, ser. FoSER '10. New York, NY, USA: ACM, 2010, pp. 81–86. [Online]. Available: http://doi.acm.org/10.1145/1882362.1882380

[11] S. Q. Lau, "Domain analysis of E-Commerce systems using Feature-Based model templates," Master's thesis, University of Waterloo, Canada, 2006.

[12] "Flagship docs 4," accessed: 2015-10-22. [Online]. Available: https://github.com/wtg/flagship_docs_4

[13] D.-I. Kang, R. Gerber, L. Golubchik, J. K. Hollingsworth, and M. Saksena, "A software synthesis tool for distributed embedded system design," *SIGPLAN Not.*, vol. 34, no. 7, May 1999.

[14] E. Andersen, S. Gulwani, and Z. Popovic, "A trace-based framework for analyzing and synthesizing educational progressions," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '13. New York, NY, USA: ACM.

[15] V. Le and S. Gulwani, "Flashextract: A framework for data extraction by examples," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM.

[16] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *16th International Conference on VLSI Design, 2003. Proceedings.* Institute of Electrical & Electronics Engineers (IEEE), 2003.

[17] I. Assayad, V. Bertin, F. x. Defaut, P. Gerner, O. QuÃľvreux, and S. Yovine, "Jahuel: A formal framework for software synthesis," in *in ICFEM'05*, 2005.

[18] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '10. New York, NY, USA: ACM, 2010.

[19] A. Solar-Lezama, "Program sketching," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5-6, Aug 2012.

[20] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *POPL'10*, Jan. 2010, pp. 313–326.

[21] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement and code transplants to specialise a c++ program to a problem class," in *Genetic Programming*. Springer Science + Business Media, 2014.

[22] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-compiler software optimization for reducing energy," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014.

[23] E. Bondarev, M. R. V. Chaudron, and E. A. de Kock, "Exploring performance trade-offs of a jpeg decoder using the deepcompass framework," in *Proceedings of WOSPÃćÂĂÂŹ07*, Feb. 2007, pp. 153–163.

[24] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya, "Archeopterix: An extendable tool for architecture optimization of aadl models," in *Proceedings of the International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, Feb. 2009, pp. 61–71.

[25] A. Martens, H. Koziolek, S. Becker, and R. H. Reussner, "Automatically improve software models for performance, reliability and cost using genetic algorithms," in *Proceedings of the 1st Int. Conf. on Performance Engineering*, Feb. 2010, pp. 105–116.

[26] M. Harman, "The current state and future of search based software engineering," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007.

[27] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.

[28] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, 2009.

[29] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, 2004.