A General Algebraic Theory and Derived Software Framework for Automated Tradeoff Analysis Tools

Chong Tang, Kevin Sullivan, Hamid Bagheri (GMU), Ke Dou
University of Virginia (and George Mason University)
Charlottesville, Virginia 22904-4740
Email: {ctang, sullivan, kd6ck}@virginia.edu
and bagherih@acm.org

***Abstract*—This paper presents a formal, general algebraic theory of design space tradeoff analysis tools, and a map-reduce-based framework, derived mechanically from the theory, for implementing such tools. The theory is organized as a hierarchy of Coq typeclasses in a style similar to that being used elsewhere to formalize abstract mathematics. From this theory, using Coq's extraction function, we produce a polymorphic framework (in Scala) that developers specialize and extend to produce domain-specific trade-off analysis tools. As a test and demonstration, we instantiated this framework using code re-engineered from an earlier, ad hoc, only partially automated system for tradeoff analysis of object-relation mappings. Our new tool reduced the time taken by one analysis from weeks to hours.**

I. Introduction

When the consequences of variations in design decisions in candidate implementations of a given specification are unclear, it can be important to conduct systematic tradeoff studies. Such studies help reveal how system properties in multiple dimensions vary across implementations, revealing how stakeholders might be impacted, and what implementations might best serve the needs of a given project.

To conduct such a study, one starts with a specification, generates many variant implementations, and applies property measurement (or estimation) functions to each implementation. The result is a *tradespace* that associates a vector of property estimates with each candidate implementation. One then analyzes the tradespace to rule out strictly suboptimal solutions, identify Pareto-optimal solutions, understand tradeoffs on the Pareto front, and ultimately select a best solution.

Tradeoff analysis can help decision-makers by revealing designs that people might miss [9], 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, even runtime.

Yet, today, systematic tradeoff analysis remains rare. We lack both science and technologies to support it. Instead, developers are usually given specifications that constrain certain choices (often function) but that leave other relevant properties unspecified (often non-functional properties such as performance, reliability, evovability). Developers are then 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-relation 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.

Single-point, heuristic methods are increasingly inadequate. Key decisions are often made early and locked in by subsequent development without a full understanding of their impact on other system properties. Consequences are felt in opportunity costs of suboptimal systems, reduced stakeholder satisfaction, trouble late in development, cancelled projects, and failed systems.

Our aim is to develop scientific foundations and general-purpose software technology for practical analysis of system properties, tradeoffs, and value. We leverage advances in several areas to start to build such a science and technology base. The main contributions of this paper are (1) a formal, general algebraic "theory" of, and (2) a mechanically derived, polymorphic, map-reduce-based, general-purpose framework for, tradeoff analysis tools, along with (3) a demonstration system built on the framework that strongly suggests that the approach works.

II. Overview of Approach and Contributions

In this section we described how our approach leverages advances in several areas of computer science and engineering.

A. Constructive Logic Proof Assistants

First, we use the enormous expressiveness of dependent type theory in modern constructive logic proof assistants to produce precise, abstract, general, computationally effective theories of domains such as tradeoff analysis. We present an algebraic theory of tradeoff analysis tools structured as a hierarchy of Coq [5] typeclasses, in a style similar to that being used by mathematicians [16], [18] to formalize hierarchies of abstract algebraic structures (e.g., groups, fields, topological spaces).

B. Certified Programming with Dependent Types

Second, from this theory, we use Coq's extraction facility to derive a certified [1] implementation of a general-purpose tool *framework* (in Scala). It is then specialized and extended with user-defined, domain-specific types and functions (and in some cases, proofs), subject to the specified laws, to create domain-specific analysis tools. The theory provides implementations of functions common to all instances and expresses laws that all instances must obey.

C. Formal Synthesis from Specifications

Third, as we [3], Dwivedi et al. [11], and others have been showing, we can increasingly synthesize many implementations from given specifications, often in large numbers, particularly for tradeoff analysis. We recently showed that we can use a relational logic model finder to exhaustively synthesize relational database schemas as well as test inputs for dynamic analysis of performance from relational logic specifications of object-oriented data models [2], [3].

Our framework is meant to be specialized using any types of specification, implementation, and property estimation function. When specification-driven implementation synthesizers are available, they should be easy to plug in. When they are not, other hand-crafted functions can be used. To test this idea we re-engineered and extended our earlier, Alloy-based [8] ORM synthesizer to produce a fully automated, synthesis-driven framework-based ORM tradespace analysis tool.

With this tool, we are now able to fully replicate the largely manual analysis of synthesized database schemas that we reported in our earlier work [3]. The tool works with far higher reliability, and is just one of many possible specialized instance of a general, theory-based framework. We can now rapidly produce tool variants. Our tool has reduced the time required to analyze thousands of candidate solutions from weeks (involving tedious manual execution of synthesized benchmarks) to just hours.

D. Scalable "Big Data" Analytics

Fourth, we plan to use big data analytics, particularly map-reduce [7], to reduce analysis runtimes. While synthesizing spaces of solutions from specifications may not always be easily parallelized, applying independent property estimation functions to independent implementations is. The problem has a natural map-reduce structure. The use of scalable map-reduce technology can benefit many instances of our framework, so it is sensible to support it as a common middleware plug-in. We have not yet implemented this middlware, but it is on our near-term roadmap.

We can give a sense of the performance benefits we expect based on experiences with our ORM tool. For several specifications, our tool takes roughly three hours to generate roughly one thousand candidate ORM solutions. It then takes over eight hours to measure them all. Each measurement function, applied to each solution, creates a database and then runs several dynamic loads to profile time and space performance for reads and writes. On a typical Intel Core i7 PC, it takes about thirty seconds to measure each database. Using a 64-node Hadoop cluster should reduce measurement time from eight hours to somewhere between ten and twenty minutes, reducing overall runtime from about eight hours to three.

E. Relationship to Our Recent Work

This paper builds on results reported in two earlier works. In one [2] we used relational logic model finding to synthesize spaces of relational database schemas and to evaluate these schemas by applying multiple previously published *static* measurement functions. By *static* we mean that these functions predict properties based on schema structure only.

This work formulated an ORM domain-specific specification language embedded in the Alloy logic; an Alloy representation for MySQL schemas as a semantic domain; and constraints embodying a semantic mapping associating ORM specifications with Alloy representations of MySQL schemas. We showed that by solving the constraints of this mapping function, we could exhaustively generate spaces of MySQL schemas for a given specification.

In our most recent paper [3], we showed that we could also generate test loads for *dynamic* analysis of properties of MySQL solutions. Fair comparison of variant designs required that they all be benchmarked under a common application-level load, but such a load has to be specialized to the interfaces that variant schemas present to an application. We showed that implicit in Alloy representations of MySQL schemas was an abtraction function that could be used to specialize common loads to the diverse interfaces presented by different schemas derived from a given specification. We then used this technology to test the predictive power of the static measurement functions. We found them to be at best weak predictors of performance as seen in dynamic testing.

Several important questions, problems, and opportunities were left unaddressed by this previous work. How might the approach generalize? Can we formalize a general theory? How could such a theory be leveraged to yield a general-purpose framework for implementing diverse anaysis tools? Can such a framework support automation in a manner that is readily scalable using big data technologies?

This paper directly addresses these and related questions. As a test and demonstration, we use a heavily re-engineered and significantly extended version of code developed in our previous work. We do not claim novelty in synthesis in this paper. Rather, the novelty is in the production of a formal theory linked to a general-purpose tool framework, validated by the instantiation of an ORM tool instance. We present ORM tradespace analysis only as a concrete and useful example and test case, but no longer as a novel result.

III. SEPARATIONS OF CONCERNS EFFECTED BY INCOMPLETENESS IN SPECIFICATION

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.

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.

B. 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.

C. 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. We defer formal treatment of stakeholder utility to another paper.

D. 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.

E. The Evolution of Incompleteness in Design

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.

F. Examples

Consider object-relational mapping. Object-oriented data models serve as specifications for application database schemas. While these specifications constrain schemas, they are silent on such properties as performance. 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. Class diagram specifications are incomplete regarding these other properties.

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 that might not be desirable.

As a second example, consider a type of specification that defines the *invokes* relation on methods in an object-oriented system. A method, A, $invokes$ a method, B, if evaluating A in certain states results in the execution of B. There are several ways to implement an invokes relation. A can $call$ B explicitly, or A can $emit$ an event that $invokes$ B implicitly [17]. Given a specification with n $invokes$ tuples, there are 2^n ways to realize the specification using implicit and explicit invocation.

While all of them are equally good at satisfying the specification, they will differ in other properties of interest in design. Explicit calls statically couple callers to callees. These calls can be typechecked and resolved statically, but they complicate code, reduce reusability, and have other negative consequences. Implicit invocations, created dynamically by registering callbacks with events, have higher overhead and are hard to check statically, but they also avoid static coupling and can greatly improve modularity, reconfigurability, and evolvability. A specification that includes only invokes relationships and is silent on these other properties, requiring downstream activities to select combinations of implicit and explicit invocations that produce desireable overall results.

G. The Upshot of Silence in Specification

The upshot of this analysis 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. The rest of this paper presents details of one approach, linking theory to technology, for addressing the underlying shortcomings in the current state of knowledge, art, technology, and practice.

IV. A Constructive Logic Theory of Tradeoffs

We now present a formal theory of tradeoff analysis tools as a hierarchy of typeclasses in Coq. Our style is similar to that used by some mathematicians to formalize algebraic hierarchies for computer verification of proofs of theorems. We introduction such uses of Coq and then we explain how we borrow from this work to define a hierarchy of algebraic structures expressing a certain theory of tradeoff analysis tools.

A. *Typeclass Hierarchies for Formalizing Mathematics*

Recent years have seen major breakthroughs in the use of Coq and other proof assistant technologies for research-level mathematics. Results include Gonthier's computer-verified proofs (using Coq) of both the four-color theorem (2005) and the Feit-Thompson theorem (2012), which states that every finite group of odd order is solvable [12], [13]; and Hales' (2014) computer-verified proof (using Isabelle and HOL Light) of Kepler's 400 year old conjecture on the efficiency of pyramidal packing of spheres [10]. Hales' previously informal proof was three hundred pages long and took a team of twelve mathematicians four years to verify by hand to an estimated 99% level of confidence. The Fields Medalist Voevodsky is now using a version of Coq, with an international team of mathematicians and type theorists, to rebuild the foundations of mathematics on homotopy type theory. This work is based on insights into deep connections between types in constructive logic and topology [18].

The work we report in this paper is based on work that formalizes the standard algebraic hierarchy of monoids, groups, rings, fields, categories, functors, and so forth, and standard instances (such as the integers, rationals, exact reals, etc.) using typeclasses. One key goal in such work is to define general-purpose concepts, such as *group*, in a way that establishes a common interface to diverse instances (e.g., integers under addition). A second goal is to express and leverage the natural inheritance relationships among different structures. For example, a group inherits of the structure of a semi-group, and adds more, and can always be coerced into a semi-group by forgetting that additional structure.

Typeclasses are first-class constructs in Coq that generalize and extend the notion of typeclasses as introduced in Haskell. Abstract algebraic structures are expressed as typeclasses that define records whose fields are of types representing carrier sets, operations and relations on these carrier sets, and laws that these collections of elements must follow [16].

To formalize the concept of a *group,* for example, one might define a typeclass with four component values (whether as fields or parameters—a detail that we need not discuss here). The four values would include one, T, of sort Set (i.e., a type) representing the carrier set of elements in the group; another, of type T, specifying the neutral element of the group of the given type; a third, of type $T->T->T$, representing the binary operation of the group; and a fourth, of a type expressing the proposition that every element of has an inverse under the given operation. A value of this type would represent a proof of this proposition for a given carrier set, operation, and neutral element. Coq's dependent typechecking mechanism will prevent the instantiation of a typeclass without proper values for these elements. In particular, the representation of laws as propositional types, and the need to provide proofs as values of these types, ensures that typeclass instances always satisfy their invariants.

At bottom, Coq typeclasses support the definition of generic structures using ad hoc polymorphism. Indeed, one of the basic purposes of typeclasses in Coq, Haskell and other languages is to support overloading of operations common to a diversity of objects that share algebraic structure but that are otherwise unrelated. For example, to overload the $plus$ operation of a *group* typeclass to apply to both a group of natural numbers under addition and to a group of symmetries of a regular polygon under rotations one would simply instantiate the group typeclass with appropriate parameters defining the necessary carrier sets and operations and providing proofs of the group laws for these particular values.

B. *A Typeclass Hierarchy for Tradeoff Analysis Tools*

We formalize a hierarchy of abstract algebraic structures for trade-off analysis as a hierarchy of typeclasses linked by coercive subtyping relationships. Each typeclasses characterizes a class of possible instances in terms of fields whose values represent carrier sets, operations, and laws. At a minimum, the components of a typeclass characterize the *types* of such elements. When *values* are shared across all instances of a typeclass, a typeclass can bind such values, as well.

1) Tradespace Typeclass: Let us begin by looking at the most abstract, least structured typeclass in our hierarchy. We called it $Tradespace$. Read as a *specification*, it expresses the functionality of a broad family of tradespace analysis tools at a high level of abstraction. It can also be read as an abstract, indeed algebraic, definition of a broad class of mathematical structures: in essence, a family of tradespace analysis tools that behave in a certain manner.

```
Class Tradespace := {
    SpecificationType: Set
  ; ImplementationType: Set
  ; MeasurementFunctionSetType: Set
  ; MeasurementResultSetType: Set
  ; synthesize : SpecificationType → list (ImplementationType × MeasurementFunctionSetType)
  ; runBenchmark: ImplementationType × MeasurementFunctionSetType → (ImplementationType × MeasurementResultSetType)
  ; analyze (input: list (ImplementationType × MeasurementFunctionSetType)) : list (ImplementationType × MeasurementResultSetType) :=
        map runBenchmark input
  ; tradespace (spec: SpecificationType): list (ImplementationType × MeasurementResultSetType) :=
      analyze (synthesize spec)
}.
```

The first four lines accommodate variation in the carrier sets of $Tradespace$ tool instances. The types of these fields are the Coq sort called Set. All computational (data and function) *types* in Coq have Set as their type. (Types have types, called sorts.) The Coq type, nat, for example, represents the natural numbers; and the type of nat is Set. Thus the *type* nat can be given as the *value* of a field of type Set. In this way the idea can be expressed that a typeclass instance has the natural numbers as a carrier set.

Our structure has four carrier sets: (1) a type of input specifications for which tradespaces are to be generated; (2) a type of design implementations that could be generated from such specifications; (3) a type of measurement functions to be generated for purposes of assessing properties of implementations; and (4) a type of measurement results that the measurement functions will return. We generally expect measurement functions and results to be vector-valued, i.e., to comprise sets of finer-grained measurements and results. No values are given for these fields in the typeclass. Values (i.e., types defining carrier sets) have to be provided when the typeclass is instantiated.

These definitions set us up to specify a framework, polymorphic in all of these types, in which a function takes a specification (of whatever type is given in a particular instance), then generates a set of implementations (of whatever type is defined), a set of corresponding measurement functions (again of an arbitary type), and that then applies the functions to the implementations to produce a result—a dictionary associating implementation to computed property vectors (once again of any type). The great generality of this definition is made explicit in the lack of constraints on the types of objects than can be used in an instance of this typeclass. The remainder of the typeclass definition makes these idea precise.

The $synthesize$ component has a function type: from specifications to lists of implementation/measurement-function pairs. We intend that measurement functions (or benchmarks) provide fair performance comparisons of variant implementations. Note that the *types* in this function signature are *values* of the carrier set fields. Here we see the use of dependent types in Coq. No implementation is provided for this function in this typeclass. Rather, a function implementation (value) is required when the typeclass is instantied. The framework is thus highly flexible with respect to the mechanism to be used to generate candidate solutions from specifications.

The $runBenchmark$ component has as as its type a function that takes an implementation/measurement-function pair and returns an implementation/measurement pair. We intend that this is implemented by a procedure that runs the benchmark (measurement function) against the given implementation. Again, no implementation is bound in the typeclass; we leave it to typeclass instances to define how benchmarking actually works. (We recognize possibilities for restructuring this typeclass to represent measurement functions as function types, in which case we could actually define this function as applying a given measurement function to an implementation and returning the (implementation, result) pair. We do plan to refactor our hierarchy in light of the experience we have had in developing this work to date. For now, we leave this task as future work, as it is not essential to this report.)

The $analyze$ component of the typeclass has as its type a function that takes a list of implementation/measurement-function pairs, presumed to be generated by the $synthesize$ function, and that then maps the $runBenchmark$ function over this list to produce a list of implementation/measurement pairs. The result is the desired tradespace output. Because this functionality is common to all typeclass instances in the theory as we have defined it here, we bind an implementation (function value) in the typeclass itself.

Finally, the $tradespace$ component has as its type a function mapping a specification to a list of implementation/measurement pairs. We provide an implementation for all typeclass instances that simply composes $analyze$ and $synthesize$. When this function is applied to a specification it first generates an intermediate list of implementation/measurement-function pairs, and then runs all the measurement functions (in a map-reduce style) yielding an output list of implementation-measurement pairs, namely the desired tradespace.

2) ORM-Specific Tradespace Instance: To make these ideas concrete, we explain how we might use them to create a framework-based ORM tradeoff analysis tool. Suppose we want to compute a space-time performance tradespace for variant SQL schemas that could be used to implement the object model for a given application. We'd like to provide OO (e.g., UML or SysML) class diagrams as input specifications and get lists of schemas and corresponding benchmark results back. An instance of the $Tradespace$ typeclass for this application could be produced with the following parameters.

- SpecificationType: OO class diagram
- ImplementationType: SQL schema
- MeasurementFunctionSetType: an instrumented test harness for profiled execution of synthesized SQL scripts
- MeasurementResultSetType: a tuple of performance measures from instrumented benchmark execution
- synthesize: given a class diagram, produce a list of SQL schema / benchmark script pairs
- runBenchmark: run a profiled SQL benchmark script on a database with the given schema and return the schema-measurement pair

Note that no implementations need (or may) be given for the $analyze$ and $tradespace$ functions. Their implemenations belong to the typeclass/framework/theory, not to individual instances. We thus have a simple example of how typeclasses can capture a concept central to the notion of frameworks: that frameworks can factor out common code, leaving "hotspots" for user-supplied, instance-specific code. We also note that this particular typeclass does not require proofs of any laws.

We can fill in such hotspots in one of two ways. First, we could write complete, instance-specific data type definitions and function implementations *in Coq*, suitable for use as parameters when constructing a typeclass instance. A benefit would be that the extracted code for these definitions would be certifiably correct with respect to the Coq specification. On the other hand, we operated on the assumption that in many cases, it would be easier to write "stubbed out" datatype and function definitions in Coq, extract these stubs to Scala, and then write the actual implementation code in Scala. This is the approach that we used to implement the framework instances we describe in this paper.

The benefit of this "stub-based" approach is that we can write domain-specific code flexibly in an ordinary programming language. To produce our ORM tradeoff tools using

re-engineered code from our earlier work, we needed code interoperable with Java. Extracting stubs to Scala (which runs on the Java Virtual Machine and interoperates seamlessly with Java) and being able to write implementations that used Alloy and parts of our earlier code based was very helpful. It would have been impractical to try to port all of this code into Coq. The downside is that we lose the benefits of proof checking. The propositional content of a Coq specification is erased in extraction to ordinary code. What does remain, however, are at least explicit specifications of any laws. They are documented, but it is then left to the programmer to prove (or in practice to test and intuit) that they are satisfied.

To make these ideas clear, we present "stubbed out" ORM-specific data type and function definitions, and we show how, in Coq, we can instantiate our typeclass with these values. In cases where proofs of laws have to be provided at instance construction time, we have to be careful in defining "stubs" to ensure that actually satisfy the laws! We will see an example further in this paper.

We define four ORM-specific, but stubbed-out data types, and two stubbed-out functions (see the Coq code immediately below). $Inductive$ introduces a data type definition. Then comes the name of the data type and its type (Set). Following the := are available constructors. We provide just one constant constructor (value) for each stubbed-out type. The function definitions that follow have standard signatures and return minimal values of the required types using the values made available by the stubbed-out data type definitions. Finally, we use these types and functions as parameter values to instantiate an ORM-specific instance, $DBTradeSpace$, of the $Tradespace$ typeclass in Coq. The ability to instantiate a typeclass proves at least that its definition is consistent. Extracting this instance to Scala provides us with both the general-purpose framework framework code and stubs for the ORM-specific types and functions that we require. We just override and the stubbed-out definitions in Scala.

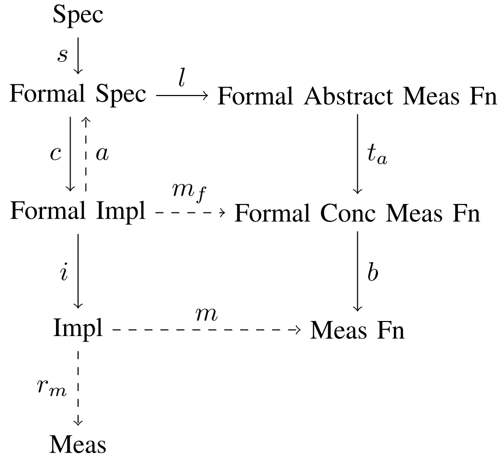
```
Inductive DBSpecification: Set := DBSpecification1.
Inductive DBImplementation: Set := DBImplementation1.
Inductive DBMeasurementFunctionSet: Set := DBMeasurementFunctionSet1.
Inductive DBMeasurementResultSet: Set := DBMeasurementResultSet1.

Definition dbSynthesize (spec: DBSpecification) : list (DBImplementation × DBMeasurementFunctionSet) :=
  (DBImplementation1, DBMeasurementFunctionSet1) :: nil.

Definition dbRunBenchmark (mfPair : DBImplementation × DBMeasurementFunctionSet) : (prod DBImplementation DBMeasurementResultSet) :=
  (DBImplementation1, DBMeasurementResultSet1).

Instance DBTradeSpace: Tradespace := {
  SpecificationType := DBSpecification
  ; ImplementationType := DBImplementation
  ; MeasurementFunctionSetType := DBMeasurementFunctionSet
  ; MeasurementResultSetType := DBMeasurementResultSet
  ; synthesize := dbSynthesize
  ; runBenchmark := dbRunBenchmark
}.
```

This approach provides (and tightly connects) both a generalized theoretical model of tradeoff analysis in a formal notation and style suitable for abstract mathematics, and an efficient, well-structured, general framework that certifiably works as expressed in the formal specification. The example is simple but it nevertheless illustrates key features of our approach.

While developing our current theory, framework, and ORM demonstration system, we repeatedly evolved the theory, regenerated framework code, and refactored our instance implementation code to conform to changes in the framework. We took as a hard constraint that, as *users* of the framework code extracted from the general-purpose typeclasses, we were not allowed to change it. We are limited to instantiating the framework just by providing necessary type and function parameter values to the framework instance constructors. We found it completely practical to keep the mathematical theory, framework, and domain-specific instance consistent. Incremental changes in the theory propagate mechanically to framework code updates, and these in turn were readily accommodated by incremental refactoring of our domain-specific code.

3) Trademaker Typeclass: The $Tradespace$ typeclass captures a very general notion of tradespace analysis, and nicely illustrates some of the aspects of our approach, but it provides far too little structure to really help the implementor of a tradespace analysis tool. We introduce a new typeclass, called $Trademaker$, to that enriches the $Tradespace$ typeclass to provide a far finer-grained, but still general, implementation architecture for tradespace analyzers.

The following diagram graphically depicts the structure of the extended typeclass, within which the more abstract $Tradespace$ is embedded. Let's first point out where the $Tradespace$ elements appear. The basic tradespace functionality takes a user-oriented specification ($Spec$ in the diagram), maps it to a dictionary (m) that associates implementations ($Impl$) with (vectors of) measurement functions ($MeasFn$), and from there produces a dictionary (r_m) that associates implementations ($Impl$) with (vectors of) measurements ($Meas$), produced by running the measurement functions against the implementations. That is the general nature of a $Tradespace$ instance.



The $Trademaker$ typeclass extends the $Tradespace$ typeclass, adding structure to define an implementation architecture for tradespace analysis tools. This extended typeclass expresses a specialized, but still very general theory of how a $Tradespace$ instances can be implemented. We sketched core elements of this implementation theory in our 2014 ICSE paper. The sketch was not formalized. There was no type theory-based formalization. The laws were not made explicit. Our software did not remotely conform to this model, as there was no general-purpose framework to instantiate. This paper extends, formalizes, and validates that early sketch, yielding a properly mathematical expression of the theory, a general-purpose implementation framework, and a proof-of-concept ORM-specific instance that recapitulates our earlier work but now in the form of but one instance of a general theory. The remainder of this section describes our implementation architecture.

The idea is that we map an end-user specification ($Spec$), such as a UML class model, to a formal specification, such as an Alloy-based object model ($Formal\ Spec$ in the diagram). The c (concretization) function maps the formal specification to a set of formal representations of implementations ($Formal\ Impl$). In our demonstration system, these are basically Alloy objects that satisfy the constraints imposed by the specification and the semantic rules mapping object models to representation of SQL schemas. The a (abstraction) function explains how each implementation represents and satisfies its specification. The l (load) function maps the same formal specification to a set of "abstract" measurement functions ($FormalAbstractMeasFn$) that will be used to produce concrete measurement function ($FormalConcMeasFn$) to produce comparable measures of the properties of the various implementations. The t_a function is responsible for specializing the common abstract measurement functions to the particular interfaces exposed by the variant implementations. The subscript a indicates that this function needs and uses the abstraction function a to do its work. The result of this process is an induced relation, m_f, that associates a (vector of) implementation-specific measurement function(s) to each implementation. The diagram commutes. The i function unparses the formal/internal representation of implementations to usable forms: e.g., Alloy solutions representing SQL schemas to actual SQL schema definitions. The b function similarly unparses formal/internal representation of implementation-specific measurement functions to useful forms: e.g., to objects that run actual SQL scripts against actual databases in order to profile their performance properties. The final result is the relation r_m that associates implementations with their corresponding property measurements.

The key idea is that users of our framework need only provide domain-specific types for the nodes in this diagram and domain-specific function implementations for the solid arcs (the other dashed-line mappings being implicit or computed). The bulk of our tool implementation effort was in producing Scala types and functions exactly in correspondence with this diagram, as required by the framework code extracted from our Coq specification. To give concrete examples, we defined a $DB\ Formal\ Spec$ class as an actual parameter for the $Formal\ Spec$ slot in this architecture. Concretely, it is a wrapper around a file handle to an Alloy specification of an object model. (Soon it will be a wrapper around a relational database entry holding such a specification.) Similarly $Impl$ is a class that wraps a file containing a MySQL schema definition. Our implementation of the c function implements our Alloy-based approach to synthesing database schemas from object models. Our other ORM-specific values are similar in their structure: classes (Scala types) wrapping representation details and function implementations that hide details of computations of the various mappings required to implement our tradespace analysis approach.

Below is the Coq typeclass that extends the $Tradespace$ typeclass to add this implementation structure, including laws that require that the diagram commutes. To produce a framework implementation we instantiate this framework in Coq with a set of stubbed-out, ORM-specific types and functions, minimally elaborated to satisfy the specified laws. Extracting this instance using recursive extraction in Coq produces stubs for the ORM-specific types and functions and a framework based on the underlying typeclasses. We elide the details of typeclass instantiation. We thus also elide the Coq scripts by which we build the proof terms needed to instantiate the framework. These details can be found in the actual Coq specification, which we will make available.

```
Class Trademaker := {
    tm_Tradespace :> Tradespace
  ; tm_ParetoFront :> ParetoFront

  ; FormalSpecificationType: Set
  ; FormalImplementationType: Set
  ; FormalAbstractMeasurementFunctionSetType: Set
  ; FormalConcreteMeasurementFunctionSetType: Set

  ; cFunction: FormalSpecificationType → list FormalImplementationType
  ; aFunction: FormalImplementationType → Formal-
```

SpecificationType
; lFunction: FormalSpecificationType → FormalAbstractMeasurementFunctionSetType
; tFunction: FormalAbstractMeasurementFunctionSetType → **list** ImplementationType → **list** FormalConcreteMeasurementFunctionSetType
; sFunction: SpecificationType → FormalSpecificationType
; iFunction: FormalImplementationType → ImplementationType
; bFunction: FormalConcreteMeasurementFunctionSetType → MeasurementFunctionSetType

; aInvertsC: ∀ (*spec*: FormalSpecificationType) (*fImpl*: FormalImplementationType), In *fImpl* (cFunction *spec*) → (*spec* = aFunction *fImpl*)

; implementationLine: ∀ (*spec*: SpecificationType) (*fImpl*: FormalImplementationType) (*impl*: ImplementationType) (*fSpec*: FormalSpecificationType),
(*fSpec* = sFunction *spec*) → (In *fImpl* (cFunction *fSpec*)) → (*impl* = iFunction *fImpl*)
→ In *impl* (map (@fst ImplementationType MeasurementFunctionSetType) (synthesize *spec*))

; testLoadsLine: ∀ (*spec*: SpecificationType) (*mfs*: MeasurementFunctionSetType) (*fCMFs*: FormalConcreteMeasurementFunctionSetType)
(*fSpec*: FormalSpecificationType) (*fAMFs*: FormalAbstractMeasurementFunctionSetType) (*impl*: ImplementationType)
(*fImpl*: FormalImplementationType),
(*fSpec* = sFunction *spec*) → (In *fImpl* (cFunction *fSpec*)) → (*fAMFs* = (lFunction *fSpec*)) →
(In *fCMFs* (tFunction *fAMFs* (map iFunction (cFunction *fSpec*)))) → (*mfs* = bFunction *fCMFs*) →
In *mfs* (map (@snd ImplementationType MeasurementFunctionSetType) (synthesize *spec*))
}.

The first two lines of the definition state that the Trademaker class extends (and is coercible to) the $Tradespace$ and $ParetoFront$ typeclasses. The latter provides structure for computing Pareto fronts of sets of vector-valued objects. Including this "mix-in" sets up the Trademaker typeclass to be futher extended to generate Pareto-optimal subsets of computed tradespaces. We elide further details of the $ParetoFront$ typeclass. This structure does illustrate the use of multiple inheritance, as employed in defining algebraic hierarchies, in which, for example, a ring extends both an abelian (additive) group and a (multiplicative) group.

The following four components provide for parameterization of typeclass instances with respect to the key *additional* carrier sets (types) of the implementation framework. Following the declarations of these type-valued parameters are seven lines that specify the dependently typed signatures of the mapping functions required to instantiate the $Trademaker$ typeclass, as illustrated in our commutative diagram. (With apologies to the reader, we note that our diagram uses abbreviations of the rather verbose type names defined in the typeclass. A careful reading will reveal the intended correspondences between these names.)

The final three components, (somewhat unintuitively) called $aInvertsC$, $implementationLine$, and $testLoadsLine$, specify *laws* that the other components of the typeclass must follow. These components are of types defined by the given logical propositions. In Coq, propositions are types (themselves of type $Prop$, as opposed to Set, which is used for computational types, whose content is preserved by extraction). 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 yield the same results: the diagram commutes. (With apologies to reviewers, the formatting of this material, which was produced by automated conversion of the Coq specification to LaTeX, will be cleaned up in a future, e.g., camera-ready, version of this paper.)

The inclusion of such propositionally-typed fields in this typeclass requires that proofs of the propositions be supplied as field values when the typeclass is instantiated. As noted previously, one in principle has the option of formalizing all parts of a tradeoff tool as a typeclass instance in Coq, in which case proofs of the laws would guarantee the integrity of a typeclass instances. In our work to date, for reasons explained above, we chose to produce a minimal, stubbed-out, instance (elided), for which producing the proofs was straightforward; and we extracted framework and framework instance code (the latter to be completed with Scala code) from these Coq constructs.

V. Evaluation

This work has shown the potential for selective use of constructive logic proof assistants to develop and present formal, algebraic "theories" of interesting families of tools, modeled on the way that hierarchies of algebraic abtractions are being formalized by mathematicians working to revolutionize the foundations of mathematics and the conduct of proof verification. As supporting evidence, we offer a simple "algebraic hierarchy" expressing both a general view of tradespace analysis, and a refined view that carries the structure of our particular implementation architecture. We leverage both the pure, dependently typed, polymorphic functional programming, and propositional-and-proof aspects of the constructive logic of Coq. Moreover, we exhibit a certified isomorphism between this theory and extracted and demonstrably useful framework code, maintained without undue effort as theory, framework, and instances evolved.

The result is, in our view, more than just an academic curiosity. This fairly simple example of an algebraic hierarchy for a family of tools embodies a significant generalization of an approach that was mostly implicit and certainly under-developed in our earlier work on dynamic ORM tradespace analysis. Moreover, the form of mathematical formalization we selected has given us the capability to mechanically develop and maintain a useful, efficient, well structured, and general-purpose implementation framework for tradeoff analysis tools. A framework instance certifiably implements the theory modulo proofs that must be discharged by human developers (unless they're willing to completely develop their implementations in Coq, which we do not expect to be practical soon, except perhaps in special cases).

To test the capability of our theory and framework to support meaningful tradeoff analysis, as we have stated, we rebuilt our previous, ad hoc tradespace analysis tool to instantiate this theoretical framework. The old code had essentially *none* of the structure expressed in our commutative diagram and our Coq specification. The tool is now completely refactored and also extended (e.g., to run measurement functions automatically!) into the types and functions required by our framework. Whereas our previous prototype software left running of dynamic tests to the manual dexterity of a graduate student, our instantiation of our new framework with these types and functions has provided us with completely automated, relational-logic-based, synthesis-driven, map-reduce-ready ORM tradeoff analysis tool.

The following three figures present visualizations of 2D projections of the 3D tradespace that we computed for one particular object-oriented data model (for an E-commerce application, the details of which are described in our 2014 ICSE paper). The points reflect performance properties and tradeoffs in this space. Pareto-optimal solutions (calculated here in 2D projections) are highlighted in red and connected by red lines. From these figures one can quickly see that there are meaningful tradeoffs to be made in such design spaces.

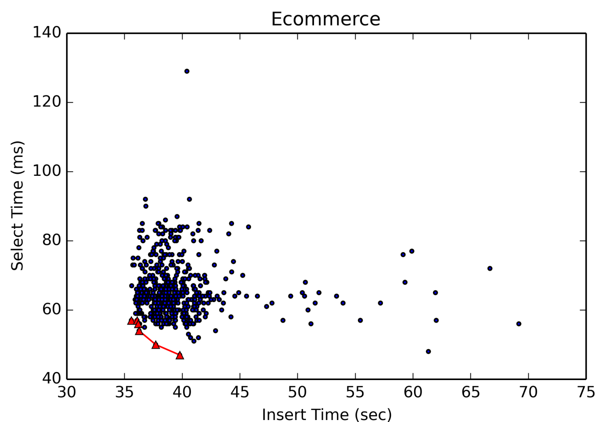


Fig. 1. My caption goes here.

We are satisfied with the results of this effort. The theory, framework, and tool instance co-evolved easily, and we now have a framework capable of supporting meaningful tradeoff analyses far beyond what we could previously carry out. In our previous work, we limited dynamic analysis to a subset of solutions predicted by static measures to be Pareto optimal. It still took several weeks of effort to carry out this analysis, as our analysis approach evolved. We now have a tool that in a matter of a few hours on a desktop PC can evaluate multiple non-functional performance characteristics of thousands of variant database designs without human assistance. We hope and expect to scale to analysis of millions of solutions in reasonable runtimes using map-reduce platforms.

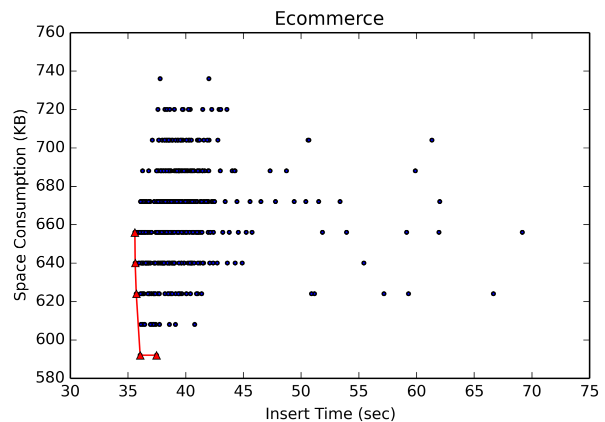


Fig. 2. My caption goes here.

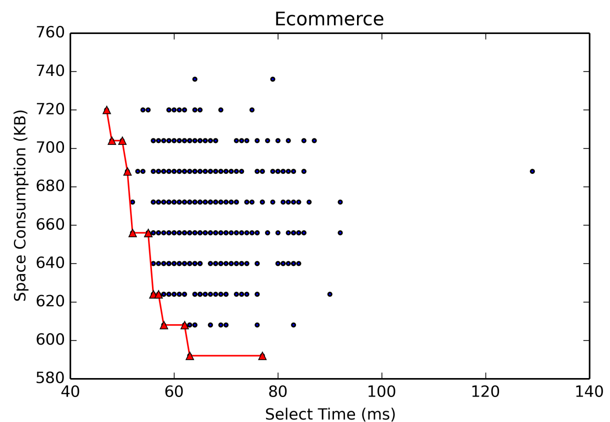


Fig. 3. My caption goes here.

We have developed a few relatively minor variants of the tool to test the flexibility of the framework in various dimensions. Our results are positive so far. For example, our first tool instance defined measurement functions as three-tuples of fine-grained functions for measuring space, read/query, and write performance of databases under relative small, formally synthesized loads. We tested the ease of extension afforded by our framework by adding measurement functions that measure the same properties using far larger, randomly generated database loads. We found that the framework accommodated the addition of measurement functions to a tradespace analysis readily. We have also taken early steps to substitute an

alternative to Alloy as a relational logic model solver [15]. It appears that this will be largely a "drop-in" substitution, as we have already validated the ability of the tool to carry out the synthesis tasks we require.

The major types in our current implementation generally wrap files stored on the file system, including, for example, many thousands of SQL schemas, XML representations of satisfying solutions to Alloy specifications, and measurement results. The modularity and abstraction required by our framework has forced our code into a state of information hiding modularity in which it will be straightforward to substitute relational database storage for the file system. This enhancement, in turn, will ease the application of scalable data management and analytics tools to tradespace data. This will include the use of Hadoop map-reduce and related methods. We are also now in the process of redesigning our web-based client software for interacting with our framework instance (including visualization, e.g., using D3.js), and producing a REST web service interface to allow our tool(s) to be called by diverse clients.

What we have not yet done is to use the framework to implement a completely different tradeoff analysis functionality—other than ORM. We are now confident, however, that the framework could easily, elegantly, and profitably support synthesis-driven tradeoff analysis functions such as those that Garlan et al. [11] are exploring in a wholly different application domain.

VI. Conclusions and Future Work

We have argued that tradespace analysis has a natural, important, indeed crucial role to play throughout the software and systems engineering life-cycle. However, our science and technology base for sysematically carrying out such analyses is very under-developed, and without good tools, such analysis is costly, tedious, and error-prone, due to the large sizes of realistic design spaces. It is thus quite rare to see software or systems engineers engaged in systematic tradeoff analysis.

This paper contributes an approach to developing, and an instance of, a formal *theory* of tradespace analysis tools, and a general software *framework*, derived from this theory, that can readily be specialized to implement diverse tradeoff analysis tools. In our experience, such a tightly coupled theory-implementation pair co-evolves readily as the theory is enhanced, sometimes as a result of insights gained during implementation-level instantiation of the tool framework.

In the future we hope to see more work of this kind: delivering not only useful software artifacts but accompanying theories expressed in ways that suitably trained mathematicians would recognize and respect as natural and appropriate. We futher hope and expect to see such theories driven to evolve as their corresponding implementations meet the demands of actual use. The result will be a virtous cycle in which formal scientific theories are driven to evolve to states of greater utility, and where such advances in theory can rapidly be turned around into theory-driven advances in software implementations. In other words, we hope that an approach such as ours can help to "close the loop" between theory and practice in software and systems engineering.

Our plans for short-term future work are evolving rapidly. We plan to transition storage of tradespace data from file system storage to a proper database. We plan then to connect such data into advanced analytics software, such as map-reduce. At the theory level, a high priority is to further extend our algebraic to another level to include stakeholders, individual stakeholder utility functions, and multi-stakeholder value reconciliation functions. Stakeholder utility functions will map the vector-valued measures for solutions to scalar utilities, which in general will differ from stakeholder to stakeholder. The value reconciliation function will convert stakeholder-indexed sets of utilities to scalar values for each candidate solution in a tradespace. These extensions will provide a high-level, formal theory for value-based, Theory-W [4] software and systems engineering.

Acknowledgments

This work was supported in part by the National Science Foundation and the Systems Engineering Research Center.

References

[1] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant,* The MIT Press, December 6, 2013.

[2] H. Bagheri, K. Sullivan, and S. H. Son, Spacemaker: Practical Formal Synthesis of Tradeoff Spaces for Object-Relational Mapping, In Proceedings of Software Engineering and Knowledge Engineering (SEKE), 2012, pp. 688-693.

[3] H. Bagheri, C. Tang, K. Sullivan, TradeMaker: Automated Dynamic Analysis of Synthesized Tradespaces, Proceedings of the International Conference on Software Engineering (ICSE'14), 2014.

[4] B. W. Boehm and R. Ross, Theory-W Software Project Management Principles and Examples, *IEEE Transactions on Software Engineering*, Volume 15 Issue 7, July 1989, pp. 902–916.

[5] Coq Development Team, *Reference Manual (version 8.4pl4)*, Inria, 2012.

[6] http://proofcafe.org/wiki/en/Coq2Scala.

[7] J. Dean and S. Ghemawat MapReduce: Simplified Data Processing on Large Clusters, OSDI, 2004.

[8] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, Cambridge, MA: MIT Press, 2006.

[9] R. P. Gabriel, Design Beyond Human Abilities, Proceedings of AOSD '06, The Fifth International Conference on Aspect-Oriented Software Design, pp. 2–2, 2006.

[10] J. Aron, "Proof confirmed of 400-year-old fruit-stacking problem," New Scientist 11:42, August 12, 2014.

[11] V. Dwivedi, D. Garlan, J. Pfeffer and B. Schmerl, Model-based Assistance for Making Time/Fidelity Trade-offs in Component Compositions, In 11th International Conference on Information Technology : New Generations (ITNG 2014), Special track on: MDCBSE: Model-Driven, Component-Based Software Engineering, Las Vegas, NV, 7-9 April 2014.

[12] G. Gonthier, "Formal ProofThe Four-Color Theorem", Notices of the American Mathematical Society 55 (11), 2008, pp. 1382–1393.

[13] R. Knies, "Six-year journey leads to proof of Feit-Thompson Theorem", http://phys.org, Oct 12, 2012.

[14] Z. Luo. Coercive subtyping. Journal of Logic and Computation, 9(1), 1999.

[15] T. Nelson, S. Saghafi , D.J. Dougherty, K. Fisler, S. Krishnamurthi, "Aluminum: principled scenario exploration through minimality," *Proceedings of the 2013 International Conference on Software Engineering*, pp. 232–241, 2013.

[16] Spitters and van der Weegen, Typeclasses for Mathematics, 2010.

[17] K. J. Sullivan, *Mediators: Easing the Design and Evolution of Integrated Systems*, Ph.D. Thesis, University of Washington, 1994.

[18] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics,* Institute for Advanced Study, June 17, 2014.