Modular Specification and Checking of Structural Dependencies

Ralf Mitschke
Technische Universität Darmstadt
Darmstadt, Germany
mitschke@st.informatik.tu-darmstadt.de

Michael Eichberg
Technische Universität Darmstadt
Darmstadt, Germany
eichberg@informatik.tu-darmstadt.de

Mira Mezini
Technische Universität Darmstadt
Darmstadt, Germany
mezini@informatik.tu-darmstadt.de

Alessandro Garcia
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br

Isela Macia
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
ibertran@inf.puc-rio.br

ABSTRACT

Checking a software's structural dependencies is a line of research on methods and tools for analyzing, modeling and checking the conformance of source code w.r.t. specifications of its intended static structure. Existing approaches have focused on the correctness of the specification, the impact of the approaches on software quality and the expressiveness of the modeling languages. However, large specifications become unmaintainable in the event of evolution without the means to modularize such specifications. We present Vespucci, a novel approach and tool that partitions a specification of the expected and allowed dependencies into a set of cohesive slices. This facilitates modular reasoning and helps individual maintenance of each slice. Our approach is suited for modeling high-level as well as detailed low-level decisions related to the static structure and combines both in a single modeling formalism. To evaluate our approach we conducted an extensive study spanning nine years of the evolution of the architecture of the object-relational mapping framework Hibernate.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.2.11 [**Software Architectures**]: Information Hiding

Keywords

Software Architectures, Modularity, Scalability, Structural Dependency Constraints, Static Analysis

AOSD'13, March 24–29, 2013, Fukuoka, Japan.

1. INTRODUCTION

A documented software architecture is an acknowledged success factor for the development of large, complex systems [29]. Traditionally, architecture description languages (ADLs) have been used to specify the architecture and verify its properties. Generally, this process has been detached from coding and the architecture specification has been considered as a means to prescribe the structure of the code resulting from programming or eventually to generate a first skeleton of that code. However, as systems evolve over time, due to new requirements or corrections, the implemented architecture starts to diverge from the intended architecture [11, 15, 22] — resulting in *architecture erosion* [26].

To combat architecture erosion, several approaches have emerged that focus on structural dependencies [10, 25, 28, 32] and whose proponents argue for automated checking of architecture specifications w.r.t. the static structure of the source code. These approaches generally allow to group[1] source code elements into *building blocks* — cohesive units of functionality in the software system — and to specify in which way a building block is allowed to statically depend on which other building block. The specification formalisms in these approaches vary and can be summarized as: (i) a flat graph with building blocks as nodes and allowed dependencies as edges [25]; (ii) a matrix notation with building blocks in rows/columns and their dependencies in the cells [28]; (iii) a graph with hierarchical nodes and component-connector style ports to manage internal/external dependencies [10]; (iv) a textual specification of access restrictions on target building blocks [32].

Such specifications are used either analytically [25] — to analyze already written code for conformance with an intended static structure — or constructively [10, 32] to enforce the code's compliance with the specification of the static structure continuously during development. Constructive approaches were proven to help developers in realizing the intended architecture. Several case studies [16, 19, 20] show that constructive approaches can prevent structural erosion [27, 33].

Though current approaches have proven to be valuable,

[1] using, e.g., regular expressions over classes or source files

they all share the property that a single monolithic specification is used and – as in case of a monolithic software system — a monolithic description of the structure does not scale and becomes unmaintainable once the software reaches a certain complexity. Sangal et al. [28] explicitly try to solve the maintainability and scalability issues using a special notation called dependency structure matrices (DSMs). However, we believe that the problem is not so much the notation. The root of the problem is the monolithic nature of the specifications. Based on some preliminary experience with modeling the architecture of real systems, such as Hibernate [5], we doubt that any such approach can scale, even with compact notations such as dependency structure matrices. As a result, typically only the highest level of components and/or libraries is considered [32, 27]; requiring different notations and tools for different levels of the design. This precludes a seamless design at various granularity levels.

In this paper, we argue that modeling a software's static structure should consist of multiple views, that focus on different parts and on different levels of detail. We take the position that like programming languages, architecture modeling languages in general should support modularity and scoping mechanisms to support modular reasoning about different architectural concerns and information hiding to facilitate evolution.

Accordingly, we propose a novel modeling approach and tool, called Vespucci, that allows to separate the specification of a software's static structure into multiple complementary views, called *slices* throughout this paper. Each slice can be reasoned over in separation. Multiple slices can express different views on the same part of the software and each slice can be evolved individually. Hence, evolution of large scale specifications consisting of several slices is facilitated by distributing work to systematically update the architecture in a modular fashion. Contrary to a monolithic specification, our approach also has the benefit that individual concerns can remain stable. Stable parts can be modularized into different slices to be separated from architectural "hot-spots", i.e., slices that require frequent changes during the evolution.

The contributions of this paper are:

- A first approach towards the specification of a software's structural dependencies that supports a modularized specification by means of individual slices.
- A new approach for modeling a software's structural dependencies that combines the advantages of hierarchical and graph-based modeling approaches to enable reasoning over a software's static structure at different abstraction levels.
- Discussion of an implementation of the proposed approach that enables the specification and checking of a software's structural dependencies.

The remainder of the paper is organized as follows. In Section 2, we briefly introduce the Hibernate framework [5], which we use to illustrate concepts of the proposed approach and to evaluate its effectiveness. Section 3 introduces Vespucci's specification language. In Section 4 we present an in-depth evaluation of Vespucci. After that, we discuss related work in Section 5. Finally, we give a summary and discuss future work.

2. ARCHITECTURE OF HIBERNATE

As part of the development of Vespucci, we did a comprehensive analysis of the architecture of the object-relational mapping framework Hibernate [5]. We provide a short overview of Hibernate and its architecture in this section since we will refer to it to discuss and motivate various features of Vespucci.

We chose Hibernate as it is a large, mature, widely-adopted software system, which has been continually updated and enhanced. We reengineered the architecture of the core of Hibernate in version 1.0.1 (July 2002) and played back its evolution until version 3.6.6 (July 2011)[2]. During this time the core grew from 2 000 methods in over 255 classes organized in 18 packages to 17 700 methods in over 1 954 classes in 100 packages.

In the following, the major building blocks of Hibernate's architecture are presented. A *building block* is a logical grouping of source code elements that provide a cohesive functionality, independent of the program's structuring, e.g., in packages or classes. The scope of a building block depends on the considered abstraction level and ranges from a few source code elements up to several hundreds. For example, Hibernate's support for different SQL dialects is represented by one top-level building block with many source code elements, but further structured into smaller building blocks for elements that abstract over the support for concrete dialects and those that actually implement the support.

Table 1: Overview of Hibernate 1.0

Top-level Building Block	2L Building Blocks	Classes contained	Elements contained	Relation to Packages
Cache	4	6	60	$\equiv$
CodeGeneratorTool	0	9	68	$\subset$
ConnectionProvider	3	5	51	$\equiv$
DatabaseActions	3	9	59	$\subset$
DataTypes	10	37	410	$\equiv$
DeprecatedLegacy	2	2	6	$\subset$
EJBSupport	0	1	22	$\equiv$
HibernateORConfiguration	2	2	39	
HibernateORMapping	12	33	389	$\equiv$
HQL (Hibernate Query Lang.)	3	9	130	$\equiv$
IdentifierGenerators	4	12	92	$\equiv$
MappingGeneratorTool	0	19	233	$\subset$
PersistenceManagement	6	35	674	
PropertySettings	0	1	43	
Proxies	0	3	23	$\subset$
SchemaTool	2	5	34	$\subset$
SessionManagement	6	10	312	
SQLDialects	3	12	119	$\equiv$
Transactions	2	4	37	$\equiv$
UserAPI	9	9	63	
UtilitiesAndExceptions	2	33	235	
XMLDatabinder	0	2	21	

The architectural model of Hibernate 1.0 consists of the 22 top-level building blocks shown in Table 1. Of these 22 top-level building blocks, 16 were further structured. In total, we identified 73 second-level building blocks. Given the size of Hibernate 1.0, we did not analyze lower levels. On

[2]Hibernate 4.0 was released after the case study.

average each top-level building block already only contains 11 classes and the 2nd level building blocks consist of even fewer classes. The key figures of the architecture are given in Table 1. In the following, we discuss those elements of the architecture that are most relevant when considering the modeling of architectures. The complete architecture can be downloaded from the project's website [2].

For Hibernate 1.0 nine of the building blocks have a one-to-one mapping to a package (cf. Table 1 – Relation to Packages $\equiv$). Six building blocks map to a subset ($\subset$) of the code of some non-cohesive package. For example, the package `cirrus.hibernate.impl` contains classes for creating proxies as well as classes related to database actions. These sets of classes have no interdependencies and belong to different building blocks. The source code elements of the remaining building blocks are spread across several packages. For example, the code related to session handling is spread across two packages in version 1.0.

Overall, the architecture features several well modularized building blocks, such as the Cache, HQL or Transactions building blocks, which are only coupled with at most three other building blocks. The number of well modularized building blocks with few dependencies is, however, small. The majority of Hibernate's functionality belongs to building blocks that exhibit high coupling, such as PersistenceManagement, SessionManagement and DataTypes.

3. THE VESPUCCI APPROACH

In this section, we first describe the three major parts Vespucci [2] consists of: (1) a declarative source code query language to overlay high-level abstractions over the source code, (2) an approach that enables the modular, evolvable, and scalable modeling of an application's structural dependencies, and (3) a runtime for checking the consistency between the modeled and the implemented dependencies. After that, we present in Section 3.4 the different modeling approaches supported by Vespucci. Finally, we discuss in Section 3.5 how the proposed approach facilitates the evolution of the specification and the underlying software and how it supports large(r) scale software systems.

3.1 High-level abstractions over source code

Vespucci is concerned with modeling and controlling structural dependencies at the code level. But, it does so at a high-level of abstraction.

Ensembles are Vespucci's representation of high-level building blocks of an application, whose structural dependencies are modeled and checked. Specifically, Vespucci's ensembles are groups of source code elements, namely type, method, and field declarations. The definition of an ensemble involves the specification of source code elements that belong to it by means of source code queries. We refer to the set of source code elements that belong to an ensemble as the *ensemble's extension*.

The visual notation of an ensemble is a box with a name label. For example, Figure 1 shows two ensembles, one called SessionManagement and one called HQL. Vespucci explicitly predefines the so-called *empty ensemble* that never matches any source elements and is depicted using a simple gray box (▢). The empty ensemble supports some common modeling tasks, e.g., to express that a utility package should not have any dependencies on the rest of the application's code.

The source code query language is introduced – mostly example-driven – in the following paragraphs. The language is not the primary focus of this paper, which is rather on modularity mechanisms for modeling structural dependencies. In fact, the approach as a whole is parameterized by the query language, in the sense that the modularization mechanisms can be reused with other more expressive query languages and more sophisticated query engines. For a more systematic definition of the current query language, the interested reader is referred to the website of the project [2].

The query language provides a set of basic predicates that can select individual fields or methods, entire classes, packages, or source files. Predicates take quoted parameters, which filter respective code elements by their signature, e.g., the predicate `package('cirrus.hibernate.helpers')` selects code elements in Hibernate's `helpers` package, using the package name as the filter. The query defines the Utilities ensemble, which we have used in modeling Hibernate's structural dependencies.

In the above example, source code elements are precisely specified by their fully qualified signature. Furthermore, wildcards ("*") can be used to abstract over individual predicate parameters. For example, the `field` predicate below selects field declarations in class `Hibernate` with any name (the second parameter is "*"), of a type that ends with the suffix `Type`. We have used the query to define an ensemble called TypeFactory which serves as a factory for Hibernate's built-in types.

```
field('*.Hibernate','*','*Type')
```

Queries can be composed using the standard set theoretic operations (union, intersection, difference), or by passing a query as an argument to a type parameter of another query. This form of composition is useful to reason over inheritance for selecting all sub-/supertypes of a given type. For example, consider the query:

```
class_with_members(subtype+('Dialect'))
```

It uses the basic predicate `class_with_members`, which selects a class and all it's members. Since the predicate expects a type to be selected, we can instead pass a subquery. The `subtype+` query returns the transitive closure of all subtypes of the class `Dialect`. Hence, the example query selects all classes (and their members) that are a subtype of the class `Dialect`. In Hibernate these represent all supported SQL dialects – the shown query actually defines the ensemble ConcreteDialects.

As already mentioned, the query language is interchangeable. What is interesting about the use of the source query language as an ingredient of our approach is that it enables modeling structural dependencies at a high-level of abstraction. Furthermore, it supports the definition of ensembles that cut across the modular structure of the code, e.g., `TypeFactory` cuts across the class-based decomposition of code. This enables feature-based control of structural dependencies.

Vespucci provides an **ensemble repository** that stores the definitions of all ensembles. It serves as a project-wide repository and provides the starting point for modeling an application's intended structural dependencies. Capturing all ensemble definitions in a single repository serves two purposes. First, it enables a model of intended structural dependencies to be modularized with the guarantee that all modules refer to the same extension for a particular ensemble. Sec-

ond, it allows modules to pose global constraints quantifying over all defined ensembles (see the discussion about global constraints in the following section).

3.2 Modeling Structural Dependencies

Dependency slices are Vespucci's mechanism to support the modularized specification of an application's structural dependencies. A slice captures one or more specific design decisions, by expressing one or more constraints over ensemble inter-dependencies, e.g., which ensemble(s) is/are allowed to use a certain other ensemble.

For illustration, Figure 1 shows an an exemplary slice, which governs dependencies to source code elements that implement the Hibernate query language, represented by the HQL ensemble. Specifically, it states that elements pertaining to HQL may be used ONLY by those pertaining to the SessionManagement ensemble. The cycle attached to the arrow pointing to HQL states that globally, i.e., for all ensembles in the ensemble repository, this is the only dependency on HQL's elements that is allowed.

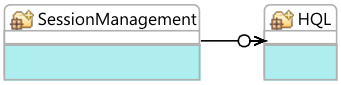

Figure 1: Dependency rule for Hibernate Query Language

Figure 2 shows another example slice, which states that source code elements pertaining to SQLDialects are only allowed to be used by PersistenceManagement's or SessionsManagement's source code elements.

Figure 2: Users of SQL Dialects

There can be an arbitrary number of slices in a model of structural dependencies; the set of ensembles referred to in different slices may overlap. Deciding about the number/kind of slices, in which one may want to break down the specification of an application's structural dependencies is a matter of modeling methodology, as we elaborate on in section 3.4. Yet, we envision the default strategy to be one in which each slice is used to express allowed and expected dependencies from the perspective of a single ensemble; this strategy was used in the case study and in the examples shown in the paper. For this purpose, the visual notation features arrow symbols that are shown next to the ensemble that is constrained[3]. For example, by looking at Figure 1 we can reason about *all* dependencies that are allowed for HQL and looking at Figure 2 we can reason about *all* dependencies that are allowed for SQLDialects.

Ensembles that participate in a slice but which have no arrow symbols next to their box are not constrained. For example, both slices refer to SessionManagement, but make no statement w.r.t. the total of its allowed dependencies. From these two slices we can see that SessionManagement's source code elements are allowed to depend on both SQLDialect's and HQL's source code elements. However, SessionManagement and PersistenceManagement are not constrained.

Constraint types are classified into two basic categories: constraints that are defined w.r.t. the allowed and those w.r.t. the not-allowed dependencies. Constraints on allowed dependencies are further classified as *Outgoing and Incoming Constraints* and *Local and Global Constraints*. The rationale for distinguishing between the above types of constraints relates to enabling modular reasoning about individual architectural concerns. Modular reasoning fosters scalability by allowing each slice to be understood as a single unit of comprehension, and also fosters evolvability as each slice can be adapted without the need to refer to other slices. We elaborate on the role that different constraint types play with these respects in the following section. Here, we exclusively focus on explaining the meaning of these different constraints.

An *incoming constraint* restricts the set of source code elements that may use the elements of a particular ensemble (target ensemble). Incoming constraints are denoted by the symbol "⇒" shown next to the target ensemble (cf. Figure 1, Figure 2). For example, the constraint in Figure 1 restricts source code dependencies, of which the target element belongs to HQL: the source of the dependency must belong to SessionManagement; source code dependencies from and to the source code elements belonging to SessionManagement are — w.r.t. that slice — unrestricted.

An *outgoing constraint* restricts the set of source code elements on which code elements of a specific ensemble (source ensemble) may depend. Outgoing constraints are visually denoted by the symbol "⇒" shown next to the source ensemble. For example, the slice in Figure 3 features two outgoing

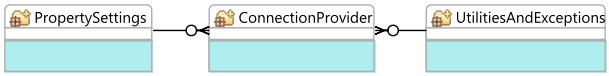

Figure 3: Constraints on the Connection Provider

constraints; from ConnectionProvider to PropertySettings, respectively to UtilitiesAndExceptions. Outgoing constraints only affect code elements of their source ensemble. Hence, the slice in Figure 3 governs the dependencies of code elements involved in providing connections (captured by the ConnectionProvider ensemble). They may only use generic functionality (captured by the UtilitiesAndExceptions ensemble), or functionality for getting and setting properties (captured by PropertySettings). The targets of the constraint (PropertySettings and UtilititiesAndExceptions) can — w.r.t. the slice in Figure 3 — depend on any other ensemble.

Global constraints quantify over all defined ensembles. Visually they are denoted by a "o" attached to a constraint. All constraints considered in the examples so far were global. For example, the constraint shown in Figure 1 affects code elements that belong to any ensemble defined in the repository of Hibernate, even if not referred to by the slice, e.g., ConnectionProvider or PropertySettings in Figure 3. Code elements of the latter ensembles are not allowed to depend on elements in HQL.

Global constraints are hard constraints w.r.t. the addition of new ensembles into the architecture. Whenever new ensembles are defined in the ensemble repository, they are included when checking a global constraint. The purpose is to provide tight control over the evolution of the archi-

[3]For this paper the visual models were compressed to save space. Hence the distinction may not be as obvious as it is when you use the Vespucci tool.

tecture. If a new ensemble has dependencies that violate a global constraint, then architects can asses whether the violation needs to be removed from the code or, whether the currently defined architectural rules are too narrow. The essential point is that an architect has assessed the situation and no uncontrolled erosion of the software's structure has occurred.

Local constraints quantify only over ensembles that are referenced in one particular slice. Visually, they are characterized by the lack of the "o" symbol. Figure 4 depicts local constraints on the implementation of Hibernate's support for different SQL dialects (e.g., "Oracle SQL","DB2 SQL"). Each dialect is realized by implementing a common interface. Elements of this interface are captured by the AbstractDialect ensemble. Support for specific dialects is captured by ConcreteDialects. The TypeNameMap ensemble captures code elements involved in implementing a specialized dictionary for mapping database type names to a common set of names. The defined constraints specify that only code pertaining to ConcreteDialects is allowed to depend on code pertaining to AbstractDialect and code in the latter is only allowed to depend on TypeNameMap's code. Furthermore, neither source code elements of AbstractDialect nor TypeNameMap are allowed to depend on elements of ConcreteDialects due to the incoming constraint between the empty ensemble and ConcreteDialects. However, the constraints of the slice in Figure 4 do not restrict in any way code elements belonging to ensembles that are not referenced by this slice, e.g., code pertaining to HQL (slice in Figure 1) could use code pertaining to ConcreteDialects.

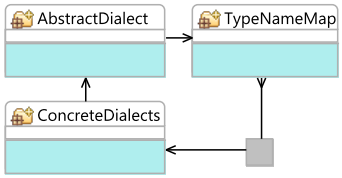

Figure 4: Supporting Multiple SQL Dialects

Local constraints provide tight control over the evolution of source code w.r.t. the scope of the ensembles referenced in a slice. Their purpose is to capture localized rules that reason only over a part of all the dependencies in the architecture, e.g., as in Figure 4 where only dependencies pertaining to the implementation of multiple SQL dialects are considered. The implementation details of involved ensembles can change (and respectively their extensions), but the changes are guaranteed to adhere to the specified allowed/expected dependencies. The rest of the architecture can evolve independently, i.e., new ensembles and dependencies can be introduced as long as they do not violate the localized rules.
Different kinds of dependencies can be constrained individually by annotating constraints. The kinds of dependencies are those that can be found in Java code (e.g., Field Read Access, Field Write Access, Inherits, Calls, Creates,...; a complete reference is available online [2]); by default, all kinds of dependencies are constrained and no further annotation of a constraint is necessary. Dependency kinds are important when documenting detailed design choices.

For example, Figure 5 restricts only dependencies of the create kind (i.e., object creations) to the ConcreteConnectionProviders; only the ConnectionProviderFactory is allowed

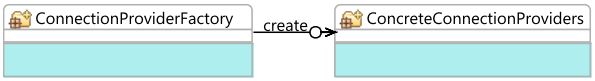

Figure 5: Restricting connection provider creation to a factory

to create new connection providers. All other dependencies are allowed for all ensembles, hence clients may use the created provider, e.g., by calling its methods. The range of possible applications is broad, e.g., one can also disallow classes from throwing particular exceptions, while allowing their methods to catch them.
Nesting of ensembles is also enabled in Vespucci to reflect part-whole relationships. The information about child/parent relationships between ensembles is stored in the global repository. For illustration consider that the slice shown in Figure 4 actually models the internal architecture of Hibernate's support for SQL dialects. One can express this relation by making the ensembles referred to in Figure 4 children of the SQLDialects ensemble, as shown in Figure 6.

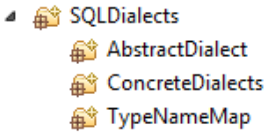

Figure 6: (Sub-)Ensembles of SQL Dialects

The extension of an ensemble that has inner ensembles is the union of the extension of its inner ensembles; i.e., an ensemble with inner ensembles does not define its own query to match source elements, but instead reuses the queries of its inner ensembles. Hence, the semantics of nesting is that constraints defined for parent ensembles implicitly apply to source code elements of all their children, e.g., constraints defined for SQLDialects in the slice in Figure 2 apply to all its children ensembles.

Constraints that cross an ensemble's border are disabled in Vespucci for keeping the semantics simple. Due to slices, this can be done without loss of expressivity. If an architect needs to define a constraint between two ensembles that do not have the same ancestor ensemble, it is always possible to specify the constraint in a new slice that just refers to the directly relevant ensembles.

With hierarchical modeling, architects can distinguish between ensembles that are involved in the architectural-level modeling of dependencies (SQLDialects) and those involved in modeling decisions at lower design levels (ensembles in Figure 4). In the following sections, we discuss how the combination of slices and hierarchies facilitates the incremental refinement of a software's architecture and is advantageous in case of software evolution.

3.3 Constraint Enforcement and Tooling

Conceptually, checking the implementation against the modeled dependencies is done as described next.

First, for each ensemble its extension along with the set of source code dependencies related to it (those that have the ensemble as source or target) is calculated; self-dependencies, i.e., source code dependencies, where the source and target elements belong to the same ensemble are filtered out. Fur-

thermore, dependencies from and to source code elements that do not belong to any ensemble are ignored.

Second, each slice is checked on its own. To do so, Vespucci iterates over all ensembles of each slice and checks that none of the dependencies between respective source code elements violates a defined constraint. For example, to check the compliance of an application's source code with the slice in Figure 3, Vespucci effectively checks that the target of all code dependencies, starting at a code element in ConnectionProvider, either belongs to PropertySettings or to UtilitiesAndExceptions.

The implementation of Vespucci's dependency checker is integrated into the Eclipse IDE. Checking is done as part of the incremental build process and incremental checking ([9]) is efficient enough for (at least) mid-sized projects such as Hibernate. Likewise the modeling side is incrementalized, hence, changes to queries are immediately reflected in the IDE.

The rationale behind the decision to ignore dependencies to source code elements that do not belong to any ensemble is that dependencies to an application's essential libraries and frameworks are most often not of architectural relevance and should not clutter the overall specification. Nevertheless, it is always possible to create an ensemble that covers some fragment of a fundamental library to restrict its usage. E.g., while it generally does not make sense to restrict the usage of the JDK, it may still be useful to restrict the usage of the `java.util.logging` API, because the project as a whole uses a different API for logging and it has to be made sure that no one accidentally uses the default logging API. One possibility to model such a decision is to create a global incoming constraint from an empty ensemble to the ensemble representing the `java.util.logging` API. However, Vespucci provides a specialized view that lists source code elements that do not belong to any ensemble to make it easily possible to find unintended holes in the specification.

3.4 On Modeling Methodology

Figure 7 schematically shows four principal ways to model the architecture of a hypothetical system consisting of four ensembles (boxes labeled 1 to 4) with Vespucci. In (A), all constraints are modeled in a single model. In (B), the model makes use of hierarchical structuring – specifically, ensembles 1 and 2 are nested into an ensemble 1&2. In (C), the model makes use of slicing; specifically, per ensemble one slice is defined, modeling only decisions related to that ensemble, but slicing at other granularity levels is conceivable (see below). In (D), the model makes use of both slices and hierarchies, which is the expected typical usage of Vespucci.

In general, the structural dependency model of a system in Vespucci consists of an arbitrary number of slices. It is a matter of modeling decisions – taken by the architect – in how many slices she breaks down the overall architectural specification. As part of this process, a trade-off is to be made between (i) creating (large(r)) slices that capture several architectural rules related to multiple ensembles that conceptually belong together and (ii) creating one slice per ensemble that just captures the architectural rules related to that ensemble. In the former case cohesiveness is fostered while in the latter case (local) comprehensibility of the architecture and evolvability of the specification is fostered.

In the Hibernate case study, as a rule of thumb, each high-level slice focused on design decisions concerning one

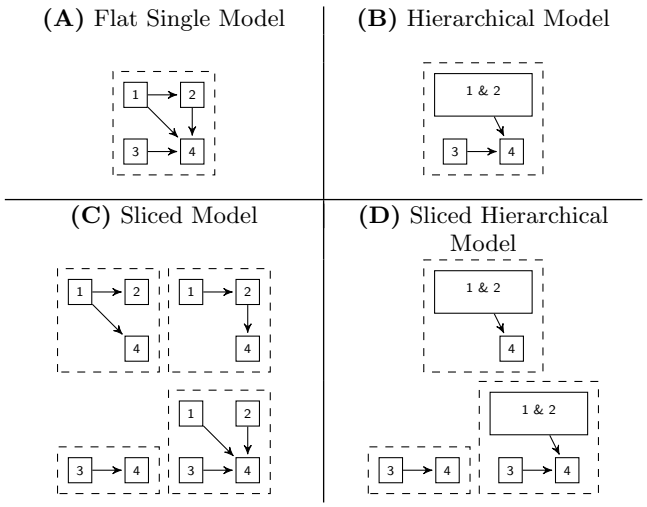

Figure 7: Alternative Architectural Models of Dependencies

ensemble. For instance, the slices in Figure 1 and Figure 3 focus on specific design decisions related exclusively to allowed *incoming dependencies* to HQL, respectively allowed *outgoing dependencies* of ConnectionProvider. Internal dependencies for ensembles with nested sub-ensembles were in general related to a small set of ensembles and hence captured in a single slice, as e.g., in Figure 4, where the internals of SQLDialects were captured.

The one-slice-per-high-level-ensemble strategy for breaking down specifications is just a first approximation. For reasons of better managing complexity and evolvability as well as understandability, it may make sense to chose more fine-grained or coarse-grained strategies. One such strategy is to split the specification of incoming and outgoing dependencies of an ensemble, if those are too complex or evolve in different ways. On the other hand, slices of related ensembles may be merged, when their separated specifications are too simple to justify separate slices or hard to understand in isolation.

One may criticize that a specification becomes complex with an increasing number of slices. However, a single specification that controls the dependencies to the same degree is no less complex and includes all information that are captured in the slices. For example, if internal dependencies are controlled, they need to be specified and maintained in a single specification as well. The focus here is to make a case for enabling the architects to break down specifications of structural dependencies in several modules that are more manageable w.r.t. scalability and evolvability and can be reasoned over in isolation. Hence, slices also facilitate distribution of work, such that large architectures can be maintained by a team rather than a single architect.

Per ensemble slicing of the dependency model may also impair understandability of dependencies pertaining to several modules. A view of the dependencies for multiple ensembles (in contrast to their individual constraints) can be advantageous for the exploration of the architecture, e.g., if one wants to follow transitive dependencies such as the path of communication from ensemble A to B. Note that if such a path is relevant to the architect, it can also be encoded as a slice. A second scenario for global comprehension is to

find all slices in which an ensemble participates. This can be supported by a simple analysis over the defined slices.

All the above said, systematically deriving guidelines for structuring architectural decisions into slices and distributing the work is a matter of performing comprehensive studies and is out of the scope of this paper.

3.5 Scalability and Evolvability

Vespucci enables architects to reason about architectural decisions concerning structural dependencies of a set of ensembles in isolation, while treating the rest of the system as a black-box, and to do so in a top-down manner. This is due to (1) Vespucci's support for breaking down the specification into slices, (2) mechanisms for expressing structural rules via a constraint system, (3) a scoping mechanism that enables to quantify locally or globally over the set of affected ensembles, and (4) Vespucci's support for enabling the hierarchical organization of specifications. The latter is a traditional mechanism to govern complexity [31] and will for this reason not be further considered in the following discussion.

Support for modular reasoning.

Slices enable the architect to focus on constraints that concern individual ensembles or a set of strongly related ensembles. This makes it possible to isolate a small set of related architectural decisions from the rest for the purpose of modularly reasoning about them, while treating the rest as a black-box.

This fosters scalability by reducing the number of ensembles and constraints that need to be considered at once: Each slice in Figure 7 (C) contains less ensembles and constraints than the model in Figure 7 (A). One may argue that slicing actually increases the overall number of elements (ensembles/constraints) — since some of them are mentioned in multiple slices. However, as they represent the same abstractions in all slices, the overall number of elements that need to be understood remains the same as in the model A.

Consider for illustration the slice depicted in Figure 2. It expresses that only SessionManagement and PersistenceManagement may use SQLDialects with the minimum amount of explicitly mentioned ensembles and constraints. No rules governing dependencies between SessionManagement and PersistenceManagement, respectively between those and other ensembles, are specified. The slice in Figure 2 models architectural constraints from the perspective of SQLDialects. Dependencies between SessionManagement and PersistenceManagement or between those and other ensembles are irrelevant from this perspective and are, thus, left unspecified. Further, we do not explicitly enumerate all ensembles that are not allowed to depend on SQLDialects.

Vespucci's constraint system for modeling dependencies and the way checking for architecture compliance operates (see previous section) is key to the conciseness of specifications. Slices are checked in isolation. The constraint system interprets the lack of a constraint in a slice as "don't care" in the sense that the presence or absence of code dependencies is ignored. E.g., potential dependencies between SessionManagement and PersistenceManagement are ignored when checking compliance with rules in slice in Figure 2. They may well be the subject of specification in other slices to be reasoned on separately.

The role played in this respect by our distinction of incoming and outgoing constraints needs to be highlighted here. It is the use of the incoming constraints in Figure 2 that enables us to talk about constraints from the perspective of SQLDialects – excluding from consideration any further dependencies in which, e.g., SessionManagement may engage. Incoming/outgoing constraints are "unilateral" – they belong to one ensemble. Without this distinction, we would be left with "bilateral" constraints; mentioning one such constraint that affects SessionManagement would require to mention all other constraints affecting SessionManagement; hence, making it impossible to slice specifications.

The ability to abstract over any dependencies that are not explicitly constrained comes in also very handy when handling ensembles that are expected to be ubiquitously used, e.g., Hibernate's Utilities ensemble. Such ensembles would typically contribute a significant amount of complexity to architectural specifications, if the specification approach requires to explicitly mention allowed dependencies. By using a constraint system this complexity can be avoided. The specification would make no mention of dependencies to Utilities, in order to leave it unconstrained.

The ability to state a constraint that affects arbitrary many ensembles without having to enumerate those explicitly is due to the ability to make global statements. Ensembles that are not explicitly mentioned in a slice are reasoned over by global constraints, e.g., the slice shown in Figure 2 implicitly states that all other ensembles mentioned in Figure 1, 2, 3, and many more, are not allowed to use SQL Dialects. This specification is much smaller compared to enumerating this fact for all other ensembles constituting the rest of Hibernate. The latter would be necessary, if Vespucci only had allowed and not-allowed constraints and no distinction between local and global scopes.

Support for evolution.

Due to slicing, architectural models also become easier to extend. First, slices remain stable in case of extensions that do not affect their ensembles/constraints. Second, affected slices are easier to identify. Finally, existing global constraints automatically apply to new ensembles.

Consider for illustration the following scenario that occurred during the evolution of Hibernate from version 1.0 to version 1.2.3. In this step, a new ensemble — called Metadata — to represent Hibernate's new support for metadata was introduced. This change was accommodated mostly incrementally. First, the specification as a whole was extended incrementally by introducing a new slice, referring to the ensembles that Metadata is allowed to use and be used from. Second, the set of existing slices that eventually required revision was restricted to those modeling the dependencies of ensembles referred to in the new Metadata slice. For example, the slice that defined constraints for DataTypes was refined to enable the usage by Metadata. Slices that modeled unrelated architectural decisions, e.g., those governing dependencies of ConnectionProvider (cf. Figure 3), did not require any reviewing. Yet, previously stated global constraints carry over to the new ensemble, ensuring e.g., that it does not unintentionally use SQLDialects (slice in Figure 2); the usage of non-constrained ensembles, e.g., Utilities, is also granted automatically.

The way the mapping between ensembles and source code is modeled has an effect on the stability of the model in face of evolution of the system. Here we hit a variant of the well-

known "fragile pointcut problem". One way to mitigate this problem is by using stable abstractions in the source code in the queries. However, this is not always feasible; in the case study queries had to be adapted as the system evolved. Here the tool support provided by Vespucci offered some help to identify changes in the source code by: a) showing elements that do not belong to an ensemble, b) showing (sub-)queries with empty results; c) specifying that a list of ensembles should be non-overlapping (i.e., to prevent accidental matches). Even so we are aware that better source code query technology and tool support for it is needed; in this paper, we focus on the modularity mechanisms on top of the query language.

4. EVALUATION

In this section, we evaluate quantitatively the effectiveness of Vespucci's mechanisms to modularize the specification of a software's intended structure. This evaluation is performed from two complementary perspectives: (a) reduction of complexity, which is measured as the number of ensembles and constraints, and (b) facilitating architecture maintainability during system evolution. As a basis we use the re-engineered architecture of Hibernate (c.f. Sec. 2), which allows us to study an architecture of a size that is representative for mid- to large-scale projects. We also give a critical discussion of the broader applicability of our results and of threats to the validity of our study at the end of the section.

The goal of our evaluation is to assess the modularization mechanisms of Vespucci and not the accuracy of architectural violation control. Therefore, even though Vespucci is targeted at continuous architecture conformance checking, the identification of violations to the architecture is not the purpose of our quantitative evaluation. Nevertheless, it is important to highlight that in terms of enforcing conformance Vespucci is able to control violations in the source code similar to related approaches [25, 21, 28, 10, 32, 3, 8].

4.1 Scalability

We first analyze the reduction in complexity when reasoning about an architecture specification. This analysis was performed by comparing the architecture of Hibernate 1.0 modeled in the four principal ways schematically depicted in Figure 7 and outlined in the previous section. The model with both slices and hierarchies (Figure 7, D) was the primary model produced during our study of Hibernate. The other three models were produced to measure the complexity reduction for the different mechanisms (hierarchies, slices, combination of both).

Scalability with regard to the number of ensembles.

We first compare different mechanisms w.r.t. the number of ensembles referenced by isolated dependency rules. The baseline is a single monolithic specification with a total of 79 ensembles, modeled by following Figure 7 (A). The other three models Figure 7 (B-D) are quantified in the diagrams in Figure 8. The y-axis of all three diagrams denominates the number of ensembles referenced per architectural model.

The diagram on the left shows reduction in complexity for hierarchical structuring only. The model is a single specification, but high-level ensembles may be collapsed to reduce the overall number of ensembles to consider at once. The x-axis denominates the number of collapsed ensembles ordered by the number of their sub-ensembles. The values on the y-axis show how many ensembles are referenced after collapsing an enclosing ensemble, i.e., the enclosing ensemble is referenced instead of all its children. The values are accumulated, since multiple ensembles can be collapsed together. For example, in a model with the top five most complex high-level ensembles collapsed, the architect has to consider 41 ensembles at once. When collapsing all ensembles in the hierarchy, we are left with 22 top level ensembles, hence hierarchical structuring reduces the number of ensembles to approx. 27% of the total (22 of 79).

The diagram in the middle of Figure 8 shows the number of ensembles per slice when using only slices (no hierarchies). The x-axis denominates the modeled slices in the decreasing complexity order (decreasing number of referenced ensembles). Almost all slices refer to less than 27% of the ensembles (12% on average). The exemption are the three first slices that capture rules for the following building blocks (of central importance) (i) persisting classes, (ii) persisting collections, and (iii) the interface to Hibernate's internal data types. The combination of both mechanisms (diagram on the right-hand side of Figure 8), yields a much smaller number of slices (x-axis), since it focuses on the top-level building blocks. In addition, the combined approach features slightly smaller slices; on average each slice references only 9% of the total number of ensembles.

Scalability with regard to the number of constraints.

In the following, we compare how much each mechanism reduces the number of constraints used in isolated dependency rules. The comparison is similar to the comparison regarding the number of ensembles and the numbers are shown in Figure 9. The x-axis is organized in the same manner as in Figure 8. The y-axis denominates the number of constraints that are referenced in each architectural model.

The y-axis for hierarchical structuring (left-hand side diagram in Figure 9) shows the total number of constraints after collapsing an enclosing ensemble. The number includes (i) constraints that are abstracted away, since they are internal to the enclosing ensemble (cf. Figure 7 B; 1&2) and (ii) constraints that are abstracted away, since several constraints at the low level are subsumed by a single constraint at the high level (cf. Figure 7 B; 1&2 to 4). Both internal and external constraints contribute approx. half of the reduction in constraints (external slightly outweighs internal). As in the evaluation for ensembles, the y-values for the hierarchical composition (B) are accumulated, since we can use several hierarchical groupings together. For the architectural models using slices (C,D) the number of constraints is simply the number of constraints modeled in one slice.

The baseline (A) consists of 705 constraints in a single specification. If we consider the hierarchical model and collapse all enclosing ensembles, approx. 2/3 of the constraints are removed (down to 214, last value in the left-hand side diagram in Figure 9)). In comparison, slices (diagram in the middle of Figure 9) show less than 5% of the total number of constraints and 1,3% on average (9 of 705) per slice. The combination of slices and hierarchical structuring (right-hand side diagram in Figure 9) features slightly smaller slices; on average 0.9% (6.5 constraints) of the total of 705 constraints modeled.

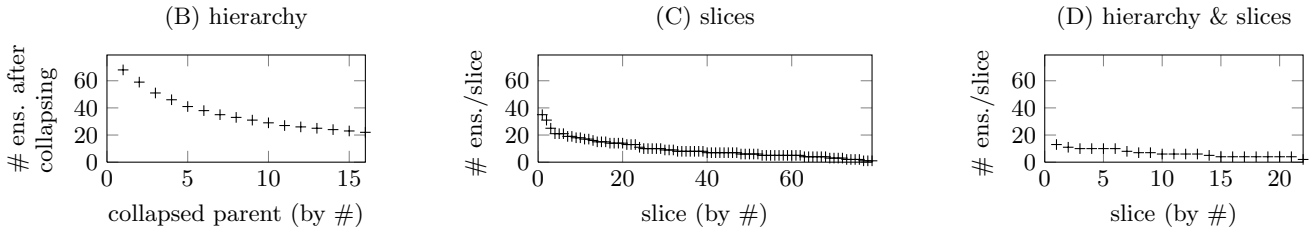

Figure 8: Comparison of ensemble reduction w.r.t. hierarchies and architectural slices (Hibernate 1.0)

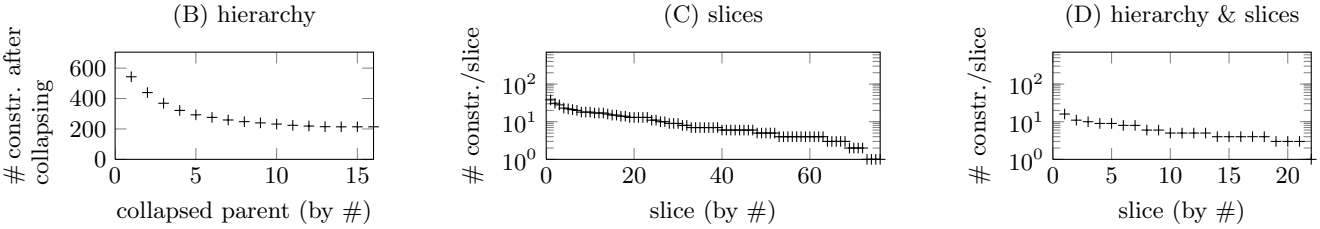

Figure 9: Comparison of constraint reduction w.r.t. hierarchies and architectural slices (Hibernate 1.0)

Scalability with regard to the number of slices.

To control the architecture of Hibernate we have modeled top-level slices comparable to Figure 7 (D) and slices for the internal constraints of the 16 ensembles that are further structured; totaling to 35 slices. Thus, the overall number of slices is smaller than the overall number of ensembles (79) and remains manageable. Note that in these models we do not use the total of the 705 constraints. First, three ensembles at the top level (SessionManagement, PersistenceManagement, and UserAPI) have no slice (and no constraints), for the reason of being used by and using almost all other ensembles. This is an inherent problem of the modularization of the software system and should be treated by refactoring the code base. Second, the modeled top-level constraints subsume several constraints on the internal ensembles. We found the control provided by the top-level constraints mostly sufficient during the evolution of Hibernate. Hence, we modeled detailed constraints only in few cases to further our understanding of the dependencies between selected ensembles.

Summary.

In this study the hierarchical structuring included 22 ensembles and 215 constraints (both approx. 1/3 of the total). Slices are much smaller; we have to collapse the first seven enclosing ensembles of the hierarchy to reduce the number of ensembles to 35, the number referenced in the most complex slice (persisting classes). Collapsing all ensembles still references 5 times more constraints than the number referenced in the slice for persisting classes. Hence, the modeling approach based on slices scales much better by reducing each slice to 9.5 ensembles and 9 constraints on average. The combination of both mechanisms produces the best results by reducing each slice to 7.1 ensembles and 6.5 constraints on average, which means that a typical slice in the Hibernate model had about 7 ensembles and 6 to 7 constraints. Thus especially the number of constraints that need to be reasoned over at once remains manageable and includes on average only 3% of the constraints of the model using hierarchical structuring, with a maximum of 16 constraints, or 7% of the constraints in the single hierarchical model.

4.2 Evolvability

To evaluate the effectiveness of Vespucci in supporting architecture evolution we have compared a single model with hierarchies (Figure 7 B) to slices with hierarchies (Figure 7 D). The results are summarized in Table 2. The first three Columns show the analyzed version, its release year, and the number of LoC as an estimate for the size. Columns four and five characterize the architecture evolution in terms of ensembles and their queries. Overall, the number of ensembles has doubled. Column six shows the total number of slices in each version. We followed the methodology of one slice per ensemble – hence, the number of slices roughly follows the number of ensembles, with the exception of those ensembles that were not constrained (c.f. Sec. 4.1). Column seven shows that on average 33% of all slices (1/3 of the architecture specification) remained stable w.r.t. the previous version. The least stable revisions were the first and the last one. In the first revision, Hibernate was close to its inception phase, hence requiring more adaptations to its features. The last revision was the most extensive in terms of the timespan covered. The last three columns compare the complexity involved in performing the required updates of the architecture specifications. Columns eight and nine show the average, resp. maximal number of ensembles per slice, whose dependencies were updated, in the approach using slicing. The last column shows how many dependencies were updated in the single hierarchical model. On average only 4% to 6% of the number of dependencies updated in the single model were reviewed per slice (the maximum ranging between 7% and 15%). This reduction in complexity of the updates per slice is comparable with the reduction of the number of constraints between (B) and (D) in Figure 9.

The numbers indicate that the maintenance of individual slices is much easier than the evolution of the single architecture model and confirm what is qualitatively discussed in the previous section.

Table 2: Analysis of the Evolution of Hibernate's Architecture

Vers.	Release Year	LoC	# Ens. (Top-Level)	Add./Rem. Ensembles	# Slices (Top-Level)	Stable Slices	Dependencies reviewed with Slices (Avg.)	Slices (Max.)	Single Model
1.0	2002	14703	22	n/a	19	n/a	n/a	n/a	n/a
1.2.3	2003	27020	26	+5 / -1	23	4 (21%)	2.4	6	61
2.0	2003	22876	28	+5 / -3	25	12 (52%)	1.9	6	40
2.1.6	2004	44404	30	+2 / -0	27	9 (36%)	2.6	6	38
3.0	2005	79248	36	+9 / -3	33	8 (30%)	4.5	8	118
3.6.6	2011	106133	39	+3 / -0	36	9 (27%)	5.0	11	87

4.3 Threats to Validity

We identify two threats to the *construct validity* of our study. First, the reverse engineering of Hibernate's architecture was primarily performed by this paper's authors, i.e., not by the original Hibernate developers. Hence, the resulting architecture design may not accurately reflect Hibernate's real/intended architecture, which may lead to inconsistencies in the results. To mitigate this threat, the architectural model was created by three people — one student, one PhD candidate and one post-doctoral researcher — that together have many years of experience on object-relational mapping frameworks. Further, we extensively studied the available documentation to make sure that the model is true to Hibernate's architecture. Yet, it is likely that a different group would reverse engineer a different architectural model. But, it is unlikely that the architecture would be such different that our evaluation would become invalid. A second threat to construct validity is that other architects may modularize the architecture specification differently, resulting in a different number and scope of slices. However, the approach that we followed — roughly creating one slice per top-level ensemble — has proven to be useful and can at least be considered as one reasonable approach.

Threats to *conclusion validity* in our study could be related to the number of ensembles and architectural constraints involved in our analysis. We tried to mitigate this threat by considering an architectural model of a significant complexity. Our analysis concerned an architectural model that involved 79 ensembles, more than 700 architectural constraints and 35 architectural slices for Hibernate 1.0.

The main issue that threatens the *external validity* of our study is that it involved a single software system. To mitigate this threat we have used a well-known medium-size framework, which has been designed by taking into consideration guidelines and good practices. These characteristics allow us to analyze the benefits of Vespucci when modeling architecture designs of well-modularized software systems. In addition, we have discussed the properties of Hibernate's architecture that influence the results and compared them to other studies. However, we are aware that more studies involving other systems should be performed in the future. All our findings should be further tested in repetitions or more controlled replications of our study.

5. RELATED WORK

Closely related to Vespucci are approaches that support checking the conformance between code and architectural constraints on static dependencies [25, 21, 28, 10, 32, 3, 8]. The key difference is that none of the above approaches (nor other related work) offers the ability to modularize the architecture description into arbitrary many slices. They rather require a self-contained monolithic specification of the architecture, which does not support the kind of black-box reasoning enabled by slices (cf. Sec. 3.5). In the following, we discuss the above approaches separately; a summary of their support for the features elaborated in Sec. 3 is presented in Table 3.

Reflexion Models (RM) [25] pioneered the idea of encoding the architecture via a declarative mapping to the source code. RM is an analytical approach that uses the modeled system architecture to generate deviations between source code and planned architecture, which is reviewed by the architect. The RM approach is not a constraint system, but rather requires the specification of the complete set of valid dependencies. Omission of dependencies is interpreted as "no dependency is allowed". Other approaches extend RM by (i) incorporating hierarchical organization [21], (ii) visual integration into the Eclipse IDE [20] and (iii) extending the process to continuously enforce compliance of structural dependencies between a planned architecture and the source code [27].

Sangal et al. [28] discuss the scalability issue of architecture descriptions and propose a hierarchical visualization method called design structure matrices(DSMs), which originates from the analysis of manufacturing processes. The key advantage is the notation (matrices) that facilitates identification of architectural layers via a predominance of dependencies in the lower triangular half of the matrix. DSM features a very verbose constraint system. For example, exemptions on lower level ensembles are encoded by the order in which rules are declared, e.g., by first allowing PersistenceManagement to use SQLDialects and then disallowing the use of ConcreteDialects. While effective, this approach requires a carefully crafted sequences of constraints.

In previous work [10] we proposed an approach to continuous structural dependency checking; integrated into an incremental build process. As in Vespucci, we referred to conceptual building blocks as ensembles. However, the specification of architectural constraints has been completely revised for Vespucci. Previously we have defined LogEn; a first order logic DSL, that integrated query language and constraint specification. However, the meaning of a violation, i.e., a constraint, is defined by the end-user, which is complex in first order logic. Hence, we provided a visual notation (VisEn), which is less complex, but focuses on documenting the architecture and hence is not a constraint system, but requires explicit modeling of all dependencies. The focus of this work was on the efficient incrementalization

Table 3: Comparison with the State of the Art

	Reflexion Models (RM) [25]	Hierarchical RM [21]	DSM [28]	LogEn/VisEn [10]	DCL [32]	Vespucci
Architectural slices	-	-	-	-	-	✓
Constraint system[1]	-	-	+	+++/-[2]	++	+++
Hierarchies	-	✓	✓	✓	-	✓
Dependency Kinds	-	-	-	-	✓	✓

[1] - (non existent) to +++ (very expressive)
[2] LogEn is very expressive; VisEn does not offer a constraint system

of the checking process, hence slicing architecture specifications into manageable modular units was not supported.

Terra et al. [32] propose a dependency constraint language (DCL) that facilitates constructive checking of constraints on dependencies; discrimination of dependencies by kind is also supported. DCL offers a textual DSL for specifying constraints. DCL's constraint system is closest to Vespucci's, and can express the not-allowed, expected and incoming constraints. Yet, it lacks outgoing constraints and a scoping mechanism such as global/local constraints, which goes hand in hand with the lack of support for slicing specifications into modular units. The language supports no inherent hierarchical structure in the architecture.

A number of commercial tools have been documented (c.f. [8]) for checking dependency among modules and classes using implementation artefacts, e.g., Hello2Morrow Sotograph [1] . However, the scope of these tools is limited; they are only able to expose violations of "certain" architectural constraints such as inter-module communication rules in a layered architecture. That is, they do not provide means for expressing system constraints.

In [3] the authors propose a technique for documenting a system's architecture in source code (based on annotations) and checking conformance of code with the intended architecture. The representation of the actual architecture in the source code is hierarchical, however, they do not support slicing of specifications in modular units and the modular architectural reasoning related to it.

Languages specialized on software constraints like SCL [17], LePUS3 [14], Intensional Views [23], PDL [24] and Semmle .QL [7] can be used to check detailed design rules e.g., related to design patterns [12]. However, they are not expressive enough for formulating architectural constraints in a way that allows to abstract over irrelevant constraints, when reasoning about a part of the architecture in isolation.

In [30] authors introduce a technique to identify modules in a program called concept analysis. A concept refers to a set of objects that deal with the same information. The authors observed that, in certain cases, there is an overlap among concept partitions. The notion of slice in Vespucci could be considered as conceptually close to the notion of concept overlapping since Vespucci supports the grouping of ensembles that are ruled by the same design decisions. Other than that slices and concepts are different in the way they are defined and used. Concepts emerge while slices are explicitly modeled. Moreover, use case slices [18] are also related to our notion of slices, but focus on the modularization of the scattered and tangled implementation of use cases.

In [13] the authors discuss foundations and tool support for software architecture evolution by means of evolution styles. Basically, an evolution style is a common pattern how software architectures evolve. This case study complements our work by helping to identify evolution styles w.r.t. a software's structural architecture. The evolvability of a software that is developed in a commercial context is also discussed by Breivold et al. [6]. They propose a model that — based on a software's architecture — evaluates the evolvability of the software. Based on our experience, the model also applies to open-source software, such as Hibernate. Aoyama [4] presents several metrics to analyze software architecture evolution. He made the general observation that discontinuous evolution emerges between certain periods of successive continuous evolution. Our case-study confirms this observation. We observed that some parts of Hibernate evolved continuously, while in other parts the evolution was disruptive. Using our modular architecture conformance checking approach architects can focus on continuous and disruptive slices individually.

6. SUMMARY AND FUTURE WORK

In this paper, we proposed and evaluated Vespucci, an approach to modular architectural modeling and conformance checking. The key distinguishing feature of Vespucci is that it enables to break down specification and checking into an arbitrary number of models, called architectural slices, each focussing on rules that govern the structural dependencies of subsets of architectural building blocks, while treating the rest of the architecture as a black box. Vespucci features an expressive constraint system to express architectural rules and also supports hierarchical structuring of architectural building blocks.

To evaluate our approach, we conducted an extensive study of the Hibernate framework, which we used as a foundation for a qualitative evaluation, highlighting the impact of Vespucci's mechanisms on managing architectural scalability and evolvability. We also quantified the degree to which Vespucci can (a) reduce the number of ensembles and constraints that need to be considered at once, and (b) facilitates architecture maintainability during system evolution. For this purpose, we played back the evolution of Hibernate's structure. The results confirm that Vespucci's is indeed effective in managing complexity and evolution of large-scale architecture specifications. However, given that we have only done one extensive case study so far, we need to carry out further case studies before final conclusions on the scalability of the approach can be made.

In future work, we will explore how IDE support can help to "virtually merge" slices into a virtual global architectural

model and to automatically create "on-demand" slices to help architects to plan a software's evolution. Obviously, further empirical studies are needed to better understand the benefits and limitations of Vespucci. New studies need to be designed to asses the impact of the approach on architect's productivity and on software quality. In this respect it would be interesting to study the effect of modularization w.r.t. enlarged control, i.e., the modularization allows to efficiently maintain an architecture containing more ensembles, which provides tighter control over the source code.

7. REFERENCES

[1] Hello2Morrow Sotograph. http://www.hello2morrow.com/products/sotograph (accessed Oct. 2012).

[2] Vespucci. http://www.opal-project.de/vespucci_project.

[3] M. Abi-Antoun and J. Aldrich. Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. OOPSLA, 2009.

[4] M. Aoyama. Metrics and analysis of software architecture evolution with discontinuity. IWPSE, 2002.

[5] C. Bauer and G. King. *Hibernate in Action*. Manning Publications Co., 2004.

[6] H. Breivold, I. Crnkovic, and P. Eriksson. Analyzing software evolvability. COMPSAC, 2008.

[7] O. de Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble. .QL: Object-Oriented Queries Made Easy. In *Generative and Transformational Techniques in Software Engineering II*. Springer-Verlag, 2008.

[8] L. de Silva and D. Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1), 2012.

[9] M. Eichberg, M. Kahl, D. Saha, M. Mezini, and K. Ostermann. Automatic incrementalization of prolog based static analyses. PADL. 2007.

[10] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. ICSE, 2008.

[11] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1), 2001.

[12] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] D. Garlan, J. Barnes, B. Schmerl, and O. Celiku. Evolution styles: Foundations and tool support for software architecture evolution. WICSA/ECSA, 2009.

[14] E. Gasparis, J. Nicholson, and A. H. Eden. Lepus3: An object-oriented design description language. Diagrams, 2008.

[15] M. W. Godfrey and E. H. S. Lee. Secrets from the monster: Extracting mozilla's software architecture. *COSET*, 2000.

[16] S. Herold. Checking architectural compliance in component-based systems. SAC, 2010.

[17] D. Hou and H. J. Hoover. Using scl to specify and check design intent in source code. *IEEE Trans. Softw. Eng.*, 32(6), 2006.

[18] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2004.

[19] J. Knodel, D. Muthig, U. Haury, and G. Meier. Architecture compliance checking - experiences from successful technology transfer to industry. CSMR, 2008.

[20] J. Knodel, D. Muthig, M. Naab, and M. Lindvall. Static evaluation of software architectures. CSMR, 2006.

[21] R. Koschke and D. Simon. Hierarchical reflexion models. WCRE, 2003.

[22] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Manage. Sci.*, 52, 2006.

[23] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views. *Comput. Lang. Syst. Struct.*, 32(2-3), 2006.

[24] C. Morgan, K. De Volder, and E. Wohlstadter. A static aspect language for checking design rules. AOSD, 2007.

[25] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. *SIGSOFT Softw. Eng. Notes*, 20, 1995.

[26] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4), Oct. 1992.

[27] J. Rosik, A. Le Gear, J. Buckley, and M. Ali Babar. An industrial case study of architecture conformance. ESEM, 2008.

[28] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. OOPSLA, 2005.

[29] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, USA, 1996.

[30] M. Siff and T. Reps. Identifying modules via concept analysis. 25(6):749–768, 1999.

[31] H. A. Simon. The architecture of complexity. In *Proceedings of the APS*, 1962.

[32] R. Terra and M. T. Valente. A dependency constraint language to manage object-oriented software architectures. *Softw.: Practice and Experience*, 39(12), 2009.

[33] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. ICSE, 2011.