

# Developer Refinement of Runtime Architectural Structure

Marwan Abi-Antoun Talia Selitsky  
Department of Computer Science  
Wayne State University  
{mabiantoun, talias}@wayne.edu

Thomas LaToza  
Institute for Software Research  
Carnegie Mellon University  
tlatoya@cs.cmu.edu

## ABSTRACT

Runtime architecture is important because it helps reason about quality attributes such as performance and security. We conducted an on-site field study to help us understand how developers understand object relationships, and what tool features a developer might need to convey their mental model of object relationships.

The subject of our study was an experienced, professional programmer. We asked the developer questions to check if he understood which tiers objects belonged to, and whether certain objects were conceptually part of other objects. We also took an initial runtime architecture and refined it to convey his intent. The developer's mental model seemed to agree with the one supported by the current, batch-oriented approach for extracting architectures.

This indicates that an interactive tool that allows making an object conceptually part of another, and abstracting away a low-level object would be useful to iteratively refine an initial object model.

## Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures

## General Terms

Documentation, Experimentation

## 1. INTRODUCTION

Over the last 20 years, the discipline of software architecture has emerged as a fundamental way to capture the high-level structure of a software system [24, 29]. A software architecture highlights how the system's components are organized and how they interact.

There are multiple ways of looking at a system's structure, called architectural views [10]. Different views have different purposes. For example, a code view shows modules as groups of source code functions, files, classes, and packages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SHARK'10 May 2, 2010, Cape Town, South Africa

Copyright 2010 ACM 978-1-60558-967-1/10/05 ...\$10.00.

Code views are useful for reasoning about dependencies between source code modules.

Runtime views show components as sets of objects and data structures. Runtime views capture important aspects of software execution more directly than code views, and are particularly useful for reasoning about runtime quality attributes such as performance, reliability, and security. Reasoning about some of these qualities requires considering the worst case as well as the average case. A *sound* architecture shows all possible communication between components, enabling worst-case analysis.

There is a good deal of work in the area of architectural extraction, which reverse-engineers high-level architectural views of a system. Architectural extraction is also known as *architectural recovery*, *architectural reconstruction*, *reverse architecting* or *architectural discovery* [19, 12]. Unfortunately, getting a sound runtime view of an architecture is still an open problem.

Even if the runtime architecture of a system is documented, it is likely to become out of date (and therefore unsound) as the system evolves. In many cases, the runtime architecture was never documented in the first place. For very small systems, an engineer can get a good idea of a system's runtime structure by reading the code. However, this is infeasible for most business-critical software, which is too large and complex for a single engineer to comprehend. Alternatively, he could use a number of existing tools that extract code views [26] but shed little light on the runtime view, or extract only partial views from a running system [27].

Instead of reading the code, it would be ideal to have a tool that could assist the engineer in extracting a sound runtime architectural view. Recent work has shown that the sound, static extraction of runtime architectures from existing object-oriented systems is technically feasible [3].

In previous work, we conducted a field study [1, 2], where the goal was to evaluate the feasibility of the approach on real production code. Our contribution in this paper is to identify how experienced professional developers think about object relationships, and compare their mental model to what our runtime architectural model supports.

**Outline.** This paper is organized as follows: In Section 2, we discuss our model for object-oriented runtime architecture. Next, in Section 3, we describe the study's method. In Section 4, we describe our results. In Section 5, we further interpret the results into requirements for interactive refinement, and discuss experimental validity and limitations. Finally, we review related work in Section 6 and conclude.

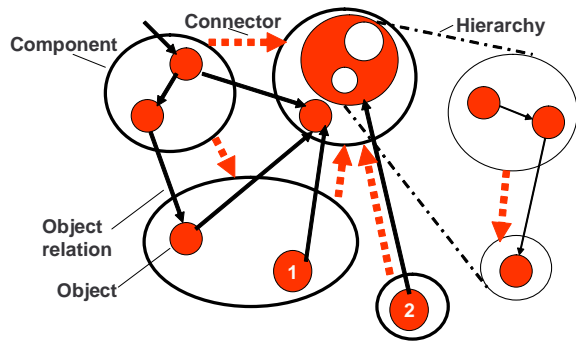


Figure 1: Architectural abstraction.

## 2. BACKGROUND

### 2.1 Architectural Extraction

Abi-Antoun and Aldrich [3] proposed the SCHOLIA approach to reason about runtime architectural hierarchy through the use of ownership types, which must be added to the code in the form of annotations before analysis begins. In order to extract a sound architecture, the annotations must be consistent with each other and with the code. The developer must run an ownership typechecking algorithm on the annotated code, and fix any high-priority warnings.

The annotations specify, within the code, object encapsulation, logical containment and architectural tiers, which are not explicit constructs in general-purpose programming languages. An *ownership domain* is a conceptual group of objects with an explicit name and explicit policies that govern how a domain can reference objects in other domains [7]. Each object is assigned to a single ownership domain that does not change at runtime. A developer indicates the domain of an object by annotating each reference to that object in the program.

SCHOLIA adopts the *extract-abstract-present* strategy [20] for architectural extraction. A static analysis *extracts* a *sound* object graph from the annotated code, using ownership to generate a containment hierarchy of objects. Next, another analysis *abstracts* the extracted object graph into a runtime architecture showing components and connectors; this step might, for example, coalesce multiple objects into a conceptual connector between components. Finally, SCHOLIA *presents* the built runtime architecture in an architecture description language (ADL).

### 2.2 Architectural Model

An Ownership Object Graph (OOG) provides architectural abstraction primarily by ownership hierarchy, by pushing low-level objects underneath more architectural objects. Thus, only architecturally relevant objects appear at the top level. In turn, each one of those objects has nested domains and objects that represent its substructure, and so on, until low-level, less architecturally relevant objects are reached.

Our current visualization (Fig. 2) uses box nesting to indicate containment of objects inside tiers, and tiers inside objects. Dashed-border white-filled boxes represent domains. Solid-filled boxes represent objects. Solid edges represent field references. An object labeled `obj:T` indicates an object reference `obj` of type `T`, which we then refer to either as “object `obj`” or as “`T` object”, meaning “an instance of the `T` class”.

Drawing from research in architectural description languages (ADLs) [21, 28], we define our runtime object-oriented architecture model as follows (Fig. 1):

**A component represents a set of objects:** a component in an architectural model represents a set of one or more objects in the runtime system.

**A connection represents relationships:** a connection in an architectural model represents a relationship between components such as points-to (an object in the source component points to an object in the target component) or calls. Connections may also include objects in their representation; for example, a pipe connector may have a buffer object inside it, and connects components A and B if an object in A writes to the buffer and one in B reads from it.

**A tier groups components:** our model supports the concept of a tier, as do many ADLs [11, 33, 6].

**Hierarchical decomposition:** in order to provide both high-level abstraction and detail, any scalable approach to documenting a software architecture must support hierarchical decomposition. In this case, components in the top-level architecture can have their own sub-architectures defined, and so forth.

**Soundness:** every object in any execution of the runtime system can be mapped to exactly one component in the architecture. Given that mapping, any runtime relationship (points-to, calls, or other relationships, depending on the intent of the diagram) between objects that map to different components is shown by some connection in the architecture.

**Traceability:** each component and connection in an architectural model should be traceable to the lines of code that are responsible for creating the objects in the component, or generating the relationship abstracted by the connection.

**Precision:** ideally, the extracted architecture should have no more connections than soundness requires. However, like any sound static analysis, the extracted architecture may have false positives—that is, the architecture may show connections that do not correspond to any runtime behavior.

### 2.3 Limitations

The most significant limitation of SCHOLIA is the high overhead associated with inserting ownership annotations into the source code, then refining them, to get them to typecheck and to ensure that they capture conceptual object containment in a way that results in the desired extracted architecture. In fact, there are still warnings remaining in most of the large systems we annotated, due to some expressiveness challenges in the underlying type system.

Another issue is that the process of refining the extracted architecture is awkward: the architect must notice and analyze architectural anomalies, identify the cases where the cause is an incorrect ownership relationship in the source code, change the ownership annotations consistently to reflect the corrected ownership relationship, and regenerate the architecture. Because this process of iterating the architecture and the annotations can be tedious and time-consuming, we wanted to spare an external developer the trouble of dealing with the annotations.

### 2.4 Research Question

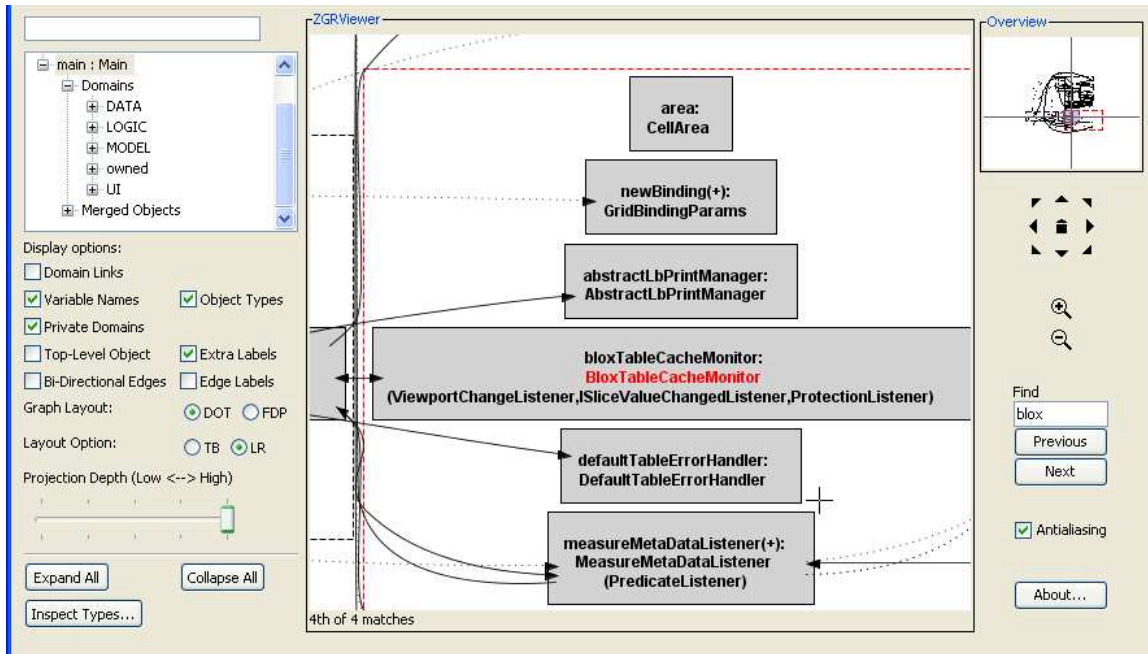


Figure 2: Architectural viewer tool.

The goal of the field study was to understand if a developer’s mental model of object relationships is congruent with one that our approach supports. We also wanted to identify the requirements for an interactive tool that enables an architect to refine an initial extracted architecture by direct manipulation, to reflect more concretely his architectural intent.

## 2.5 Summary of Previous Study

A field study is a generally accepted research method to evaluate how well a software tool or method works with real code and users [18].

During the development of SCHOLIA, we have been working closely with one of our industrial partners, LogicBlox Inc., to evaluate the technology in the field.

LogicBlox has under development a commercial system, part of which consists of over 250 KLOC of Java code. The Java code base proved suitable for study since it adopts modern programming practices such as Java generics. Moreover, the developers want to be involved in the architectural extraction because they are interested in the outcome.

During Phase I of the study [1, 2], we selected a 30-KLOC module of the system, which we will refer to as LbGrid. We then communicated with the original developers of the code to understand their architectural intent, added annotations to the code, typechecked the annotations, ran our static analysis to extract an architecture, showed hard-copy snapshots to the developers and incorporated their feedback.

During Phase I, the main focus was adding the annotations to the LbGrid module. However, we also realized that the developer was interested in viewing an object’s sub-structure. At the time, we did not have an interactive viewer. Since then, we implemented such a tool. We also spent some time addressing some of the annotation warnings, and refining the extracted object graphs. In Phase II of the study, we followed up with the same developer, and used our interactive viewer to guide our communication, and

to better understand their design intent.

## 3. METHOD

In Phase II, we conducted a follow-up with an outside developer for 2 hours. A standalone interactive viewer was used to present the extracted architecture as an object graph. We asked questions to determine if the developer agreed with the object model we presented to him. We then conducted a free-form discussion with the developer.

### 3.1 Study Design

We did not want to introduce an additional learning curve, by requiring the developer to learn an architecture description language or component-and-connector views. So we presented the extracted architecture as an object graph, using the same notation that we used during Phase I.

### 3.2 Participant

The participant for Phase II of the study was the same developer that we used during Phase I. The developer had 13 years of programming experience and was the lead developer of the LbGrid code base. Moreover, the developer was already familiar with UML diagrams, and had previously used a UML reverse engineering tool, MagicDraw [22].

### 3.3 Tools and Instrumentation

We used a standalone interactive viewer (Fig. 2). The viewer offers the following features (numbered below, and referred to in Section 4.2):

- **Display inheritance hierarchy (F1):** the tool can display the inheritance hierarchy of the types of the field declarations that an object merges.
- **Collapse/expand internals (F2):** the hierarchical representation of the architecture allows a developer to collapse or expand the sub-structure or the internals of a selected object.

- **Control unfolding depth (F3):** a developer can control the visible depth of the ownership tree.
- **Control object labels (F4):** each object in an extracted object graph represents at least one field or variable declaration in the program. An object might have multiple types, and the object graph uses one of those types as the label. The OOG can label objects with an optional field or variable name, and an optional type name. The type used in the label can also include a labeling type (discussed below).
- **Set additional labeling types (F5):** the object graph extraction non-deterministically selects a label for a given object  $o$  based on the name or the type of one of the references in the program that points to  $o$ . A developer can specify an optional list of labeling types for labelling objects. The tool adds the type decoration to an object’s label, if it merges at least one object of that type. For LbGrid, we added as labeling types the various listener interfaces. Based on this, the object `bloxTableCacheMonitor` is decorated with `ViewportChangeListener` in the diagram (Fig. 2);
- **Trace to code (F6):** the tool can show the list of field declarations and their types that may refer to a given object in the diagram. In addition, the developer can trace from an object in the diagram to the corresponding lines of code;
- **Navigate (F7):** the tool supports zooming in and out, panning, scrolling and other standard operations;
- **Search tree (F8):** the tool supports searching for an object in the ownership tree by type or field name;
- **Find label (F9):** the tool supports searching for an element on the diagram by its label.

### 3.4 Tasks

The experimenter prompted the developer to answer the following questions:

- Does the object instance identified by a name  $o$  or a type  $T$  belong to the tier  $D$  it is currently in?
- Is this object  $X$  in tier  $D$  conceptually part of another object  $Y$ , so that  $X$  can be pushed down underneath  $Y$ ?

The developer was also encouraged to give feedback at any time.

### 3.5 Procedure

The experimenter launched the viewer, and loaded an initial OOG. Then, he went through each top-level tier, and through each object in each tier. He asked the developer the above questions, to determine if the architecture was at the appropriate level of abstraction, and if not, what needed to be done. The experimenter recorded the changes suggested by the developer.

In contrast, during Phase I of the field study, the developer drew a detailed class diagram mostly from memory. He then went through the classes in the class diagram, explaining each one to the experimenter. In turn, the experimenter added annotations to the code, mostly manually, and refined the annotations to make them consistent with each other and with the code, and reduce the number of annotation warnings.

### 3.6 Analysis

After the site visit, we transcribed our notes. Next, we used qualitative protocol analysis. We built a list of activ-

ities we saw the participant engage in, and coded what he did using this model.

## 4. RESULTS

We first discuss the activities that the experimenter and the developer engaged in. Next we present our observations and support them with examples from LbGrid.

### 4.1 Activities

We spent 2 solid, uninterrupted hours with the developer, during which the experimenter used the interactive viewer, as the developer watched, answered the experimenter’s questions, or volunteered additional information. After going through all the objects in the diagram, the experimenter engaged the developer in a free-form discussion.

### 4.2 Observations

#### 4.2.1 Move an object between two domains.

The developer recognized when a given object needed to be moved from one top-level domain to another. We had already discussed the initial assignments of objects to domains during Phase I of the study. But we wanted to see if there were any changes.

The developer did request the ability to move around some of the core objects. For instance, he wanted to move the `Hierarchy` object to the `DATA` tier, and the `GridHierarchy` object to the `LOGIC` tier.

#### 4.2.2 Abstract away a low-level object.

The developer recognized when some objects were too low-level, and should not appear in a top-level domain. In contrast to the previous observation, the developer did not specify in which domain the low-level object belonged.

Our approach readily supports this activity. An OOG provides architectural abstraction primarily by ownership hierarchy, by pushing low-level objects underneath more architectural objects. Thus, only architecturally relevant objects appear at the top level.

#### 4.2.3 Make an object conceptually part of another.

The developer identified several objects that needed to be conceptually part of an object that is more architecturally relevant. For instance, he wanted `PredAggInfo`, `TypeAggInfo`, `PredicateAggInfo`, `MutliAggInfo` to be moved into the `PivotManager` object. The annotations allow an object to be only conceptually part of another object, without restricting access to the contained object.

#### 4.2.4 Recognize that an edge is missing.

Even though the diagram had many edges, the developer was able to recognize one instance of a missing edge, namely one from `TileVariableManager` to `TileVariable`.

After the study, we investigated this missing edge and realized it was due to a missing annotation on a field declaration in the program. There was also a warning that was ignored in the process of adding the annotations. This was to be expected since our approach guarantees the diagram’s soundness only if the program is fully annotated, and the annotations typecheck without warnings.

#### 4.2.5 Recognize an object that is missing.

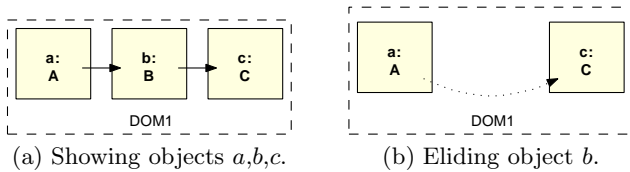


Figure 3: Example of a *summary edge*.

Although the diagram had many objects, the developer was able to recognize by visual inspection one instance of a missing object. In particular, he was expecting to see an instance of an all important, singleton `Workspace` object.

After the study, we investigated this finding and realized that the object was missing because it was created in code outside of the `LbGrid` module. However, this could be remedied using the current approach by using an annotation that indicates a “virtual” object allocation for the `Workspace` object. The notion of “virtual” or “ghost” fields is supported by many static analysis tools, to compensate for weaknesses of the analysis.

#### 4.2.6 Collapse several objects of related types.

The developer wanted us to collapse into one, several related instances of type `RowPanel`, `ColumnPanel`, and `SlicePanel`. All these types share a common supertype.

Our approach fully supports this activity. An object graph can provide abstraction by types, and merge objects in each domain based on their declared types in the program. This additional merging uses the notion of subtyping and requires the developer to specify the architecturally relevant types.

However, the current approach and tools lack a way of applying such an abstraction by direct manipulation of the object graph. Currently, developers follow a process of trial and error to select the architectural types: they select the types from a UML class diagram, apply abstraction by types, examine the result, check that the OOG merges the desired objects, but does not suffer from excessive merging, and based on a visual inspection of the output, refine the types and start over again. A more interactive approach could allow a developer to group objects of related types by direct manipulation of the object graph, for example.

#### 4.2.7 Suppress objects from the architecture.

The developer identified several objects that he did not want to see in the diagram, because they correspond to experimental, partially-implemented code.

Our approach does not directly support selecting an object and hiding it at will. Simply eliding objects could introduce unsoundness into the diagram, if the object is responsible for transitive communication between other objects visible in the diagram. Our approach does, however, support soundly summarizing an object, by adding summary edges to account for possible communication through the elided objects (Fig. 3).

Summary edges must be used sparingly, however. In some cases, adding summary edges could make the diagram imprecise. Indeed, many of these summary edges could be false positives, i.e., they may never occur during any program execution. Therefore, this approach works best when eliding only a few objects from the object graph.

#### 4.2.8 Control an object’s label.

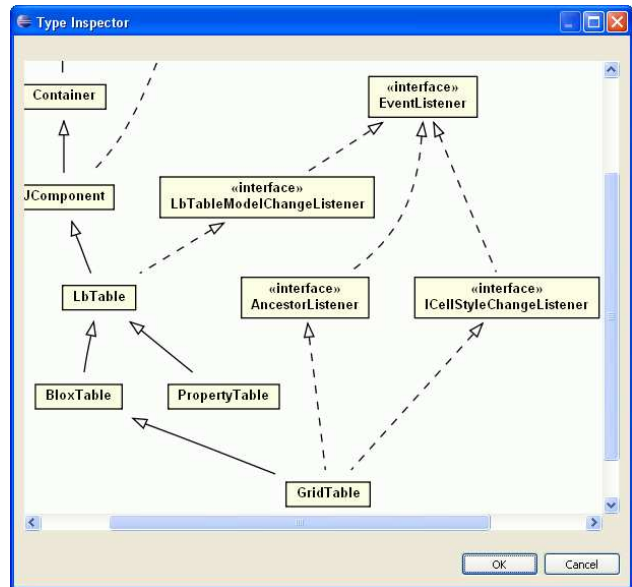


Figure 4: Inspect types dialog displayed when the user selects the `LbTable` object in the OOG.

The object graph extraction non-deterministically selects a label for a given object  $o$  based on the name or the type of one of the references in the program that points to  $o$ .

The tool has a feature (F5) to control the labeling of objects in the diagram. However, it seemed that the developer wanted an even finer-grained control over individual labels of objects. Ultimately, the tool may need to enable a developer to set the labels manually.

#### 4.2.9 Split an object into two.

The developer wanted to split one runtime object into two components. For example, in `LbGrid`, the class `AggManager` extends the class `PivotManager` using inheritance, and adds pivoting functionality. The developer wanted to see aggregation on the diagram, i.e., two separate components labeled `AggManager` and `PivotManager`.

Effectively, the developer is asking to split one runtime object into two boxes in the diagram. This violates one of the important technical properties of the OOG, namely, *object soundness*, which stipulates: each runtime object has exactly one unique representative in the object graph.

As a result, the developer’s request may not be directly achievable using our approach. Of course, the code could be refactored to represent this design intent more concretely.

### 4.3 Free-form discussion

In this section, we mention the most interesting points that the developer brought up during the free-form discussion.

#### 4.3.1 Inheritance hierarchy

The developer was very enthusiastic about the feature of the tool which can display a partial UML diagram of the type hierarchy of a given object in the object graph (Fig. 4).

The developer was already familiar with UML tools that enable you to double-click on a class in the diagram, and go to the code. He suggested a similar feature for the OOG, i.e., double-click on a type in a class diagram to go to the corresponding objects in the OOG. However, there could be

several instances of a given type in an OOG, at different levels of the hierarchy. While a class diagram would show a single box for container classes such as `Vector` or `ArrayList`, each instance of `Vector` or `ArrayList` could serve a conceptually different purpose in the design.

The tool has a related feature already available: the developer can search for an object by name or type in the ownership tree (F8), and double-click on the object to go to the object on the diagram.

### 4.3.2 Traceability to code

The developer found the feature of the tool which allows tracing from an element in the diagram to the code particularly useful. During the study, we used the feature several times to understand why a particular object appeared in a given tier.

### 4.3.3 Challenges of hierarchical representation

The developer liked the hierarchical representation of the object graph which allows expanding and collapsing objects to control the level of detail. He also correctly pointed out an inherent limitation; when all the objects of interest are at the same level, things are simple. When some of the objects of interest are nested deep, the visualization must auto-expand the hierarchy to show all the objects of interest.

### 4.3.4 Sound vs. task-specific view

Based on our discussion with the developer, it seems that he tends to favor a somewhat unprincipled approach, i.e., the ability to select and elide any object or domain in the extracted architecture (without even generating summary edges). Such a diagram, however, would be unsound, and soundness is an important technical property that our static analysis aims to achieve. Of course, soundness requires that the program's whole source code and portions of external libraries that are in use have annotations that typecheck. Moreover, the effects of reflection, dynamic code loading, or native calls, are summarized using "virtual field" annotations.

Our architectural extraction, however, is based on a principled approach. The main way to "hide" an object from the diagram is to push it underneath a more architecturally relevant one. This may require either changing the annotations, or changing the code in the case that the code does not support the desired annotation. Admittedly, this can get tedious.

In future work, we will examine the tension between having the architecture reflect everything, or only the objects of interest to a developer. Recent results seem to indicate that developers favor task-specific, potentially unsound views over global views of the entire system [9].

### 4.3.5 Showing shared or immutable objects

In the current approach, objects such as `Fonts`, `Colors`, `Strings`, etc., are treated as shared, globally aliased references. By default, an OOG does not display all these objects, since they often add needless clutter to the diagram. However, the annotations do not currently distinguish between globally aliased, i.e., `shared` objects, and objects that are truly `immutable`.

The developer looked for instances of important classes in the diagram, such as `TableCellStyle` and `CellStyle`, but could not find them. In future work, we will extend the

type system to have the `immutable` annotation, in order to be able to display such objects.

### 4.3.6 Benefits of the approach

The developer was interested in learning more about the benefits of adding annotations and extracting architectures. Indeed, evaluating the quality of an extracted architecture is subject to debate, with no generally accepted evaluation criteria. One popular approach used to evaluate an extracted architecture is to compare it to a documented, target or reference architecture. This is a popular approach. Ducasse and Pollet [12, Table 1] list 34 approaches to extract architecture, and indicate that 12 of them are used to analyze conformance between an implementation-level and a conceptual architecture.

We showed him how SCHOLIA can analyze conformance, but on a different system, Aphyds [3]. For LbGrid, however, the developer did not provide us with a designed runtime architecture. So we could not analyze the conformance of the implementation to the target architecture.

### 4.3.7 Enforcement of architectural structure

The developer suggested that the tool could generate alerts immediately when someone introduces undesired dependencies in the code. Of course, this requires knowledge about what constitutes a good or a bad dependency.

The current approach supports the above activity in the following ways [5]. First, one could use another layer of annotations, namely, domain links [7]. Domain links express local, modular constraints on how objects in one tier can access objects in another tier. Second, if the extracted architecture is documented using an Architecture Description Language (ADL), logic predicates can enforce global constraints on the architectural structure. This would make it possible to identify code modifications that impact the global architectural structure, using an automated approach, instead of the usual code inspections.

The LbGrid code base does not yet have domain link annotations. From previous experience, domain link annotations are best added at the end, until most, if not all, annotation warnings have been addressed. We plan to add these domain links in the future.

In addition, for the field study, we extracted OOGs and did not abstract them into Component-and-Connector (C&C) views in an ADL, because we did not want to introduce yet another notation. Admittedly, one could specify structural constraints on the general graph that constitutes an OOG, but the current OOG viewer tool does not support that feature, and there is no point in re-implementing functionality that is already available in architectural modeling environments.

## 5. DISCUSSION

We further interpret our results from the question and answer session with the developer, and from the free-form discussion into future requirements for the interactive refinement of a runtime architecture. Finally we discuss experimental validity and other limitations.

### 5.1 Interactive Refinement

We propose to support the interactive refinement of an extracted architecture by having a tool with a graphical user interface that supports the following features:

- **Manipulate domains.** Architects should be able to divide an architecture into tiers of components. Correspondingly, at level of an object graph, they should be able to create new domains, merge existing domains, and move an object from one domain to another (See 4.2.1).
- **Manipulate the object hierarchy.** Architects must be able to change the way objects are grouped into components. An object may conceptually be an important part of the top-level architecture, and may need to be moved up to the top level. In a more likely scenario, an object that is initially at the top level is still conceptually a part of some other top-level object, and must be moved down in the hierarchy (See 4.2.1,4.2.2).
- **Abstract objects by their types.** A default analysis creates a component for every type of object created at one level in the architectural hierarchy. However, it is common for many types to play the same architectural role. Thus, the tool should support operations that merge two components into one (See 4.2.6), and perhaps later split a conceptual component into several components, each of which includes different types of objects.
- **Summarize objects as connectors.** We often treat connectors as mere references from one object to the next. However, often higher-level connectors are really implemented by some objects in the program: examples include buffers or streams. Thus, it may be necessary to provide an operation for treating a one or more objects as a connector. While we did not encounter this situation in this field study, it did occur in one of our previous case studies [3].

## 5.2 Limitations

The developer did not use the tool himself, but was constantly prompted by the experimenter. A better interaction would have been to use a Wizard of Oz experiment [17] as defined by human-computer interaction research. We plan to conduct such an experiment after we implement an interactive editor.

## 6. RELATED WORK

**Architectural extraction.** Many of the previous techniques in architectural extraction deal only with the code architecture, rather than the runtime architecture—in part, we believe, because extracting a code architecture is a simpler problem. A consequence of this bias is that many papers often implicitly use the term “component” to mean a “package”, “module” or a collection of classes [31], without recognizing that a code architecture is just one of several architecture views. This observation is corroborated by a recent and extensive survey of architecture extraction techniques: “Because it is complex to extract architectural components from source code, those are often simply mapped to packages or files. Even if this practice is understandable, we think it limits and overloads the term component” [12, p. 587].

With the exception of static analyses (discussed below), most static architectural extractors do not track objects precisely. Instead, they represent their structural information with respect to files, directories, packages, or classes, rather than objects. For example, they express that a class is a

part of some package, or that a package is inside some other package. SCHOLIA uses object-level containment, and makes an object part of some other object.

**Static analyses.** A number of existing approaches extract, fully automatically, various non-hierarchical object graphs [16]. While these approaches can be useful for showing object interactions, they share a fundamental scalability limitation: for programs of any size, they will produce a diagram with so many objects that, in practice, the diagram will be unusable for architectural purposes.

Many of these approaches rely on points-to analysis, a fundamental static analysis to determine the set of objects whose addresses may be stored in variables, or fields of objects. All previous points-to analyses produce non-hierarchical graphs [30, 23]. Also, most points-to analyses achieve a granularity that is no coarser than an object or a set of objects.

Shape analysis is a type of points-to analysis that summarizes possible relationships among objects at runtime in a particularly precise way [25]. Shape analyses have not been demonstrated to scale to more than a few thousands of lines of code. A shape graph consists of nodes to represent a set of objects, and edges to represent points-to relations, but it is non-hierarchical in that it shows the nodes in a graph all at the same level, and does not collapse some objects underneath other objects.

Some analyses work only *intra*-procedurally, which enables them to produce manageable shape graphs that show, for a given method, the input and the output shape graphs. However, applying the analysis to the whole program *inter*-procedurally is likely to produce a large graph that would not convey much architectural abstraction.

**Case studies.** There are several published case studies in architectural extraction, mostly on big legacy systems written in procedural languages, rather than object-oriented code. For example, a successful case study extracted the code architecture of a 30-KLOC C system [15] and a multi-MLOC C system (Linux) [8]. Moreover, the steps followed by these case studies are quite instructive, and inspired the ones we followed during our own case studies, even though they were dealing with the code architecture.

To determine the code architecture of a legacy system, Tzerpos and Holt [32] used a “hybrid” process that combines facts extracted from the code and information derived from interviewing developers. These steps include: collecting “back of the envelope” designs from project personnel; extracting raw facts from the source code; collecting naming conventions for files; clustering code artifacts based on naming conventions; creating tentative structural diagrams, and collecting the reactions of the developers to these tentative diagrams; and so on, until they converged to a code architecture. Tzerpos and Holt concluded that there is a reasonably well-defined sequence of steps to go through to extract a code architecture. The steps we undertook during the field study were somewhat similar, although we dealt with the runtime architecture, and did not use clustering.

The Apache modeling project [13] manually extracted the architecture of Apache, written in C. The architectural extraction seems to have involved ad-hoc manual techniques and many people (students enrolled in a class). The only tool used for the analysis of the source code transformed the C source code into a set of syntax highlighted and hyperlinked HTML files [14]. The authors justify not using

more advanced tools by saying that “an important amount of information needed for the conceptual architecture is not existent in the code and therefore cannot be extracted by a tool” [14]. In our approach, we use annotations to specify some of that architectural intent.

**Lab studies.** In related work, we conducted an exploratory laboratory study [4], where the participants were graduate students or recent graduates rather than experienced professional programmers. The goal of the study, however, was to identify if an OOG would be useful for a developer performing a code modification task.

## 7. CONCLUSION

We discussed a field study to evaluate an architectural extraction tool with an outside developer. The study shed some light on how experienced developers think in terms of objects and their relationships. Their mental model seemed to mostly agree with our model for runtime object-oriented architecture. The study also suggested requirements for future, more interactive tools.

## Acknowledgements

The authors thank William Scherlis, Brad Myers, and Jonathan Aldrich for their useful advice on conducting the field study. The authors also thank Molham Aref and the developers from LogicBlox Inc., for hosting the field study, answering our questions, and having the follow-up meeting.

## 8. REFERENCES

- [1] M. Abi-Antoun and J. Aldrich. A Field Study in Static Extraction of Runtime Architectures. In *PASTE*, 2008.
- [2] M. Abi-Antoun and J. Aldrich. A Field Study in Static Extraction of Runtime Architectures. Technical Report CMU-ISR-08-133, CMU, 2008. Extends [1].
- [3] M. Abi-Antoun and J. Aldrich. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *OOPSLA*, 2009.
- [4] M. Abi-Antoun, N. Ammar, and T. LaToza. Questions about Object Structure during Coding Activities. In *Cooperative and Human Aspects of Software Engineering (CHASE)*, 2010.
- [5] M. Abi-Antoun and J. Barnes. Enforcing Conformance between Security Architecture and Implementation. Technical Report CMU-ISR-09-113, CMU, 2009.
- [6] Acme. [www.cs.cmu.edu/~acme](http://www.cs.cmu.edu/~acme).
- [7] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.
- [8] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a Case Study: its Extracted Software Architecture. In *ICSE*, 1999.
- [9] M. Cherubini, G. Venolia, and R. DeLine. Building an Ecologically valid, Large-scale Diagram to Help Developers Stay Oriented in Their Code. In *VL/HCC*, 2007.
- [10] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architecture: View and Beyond*. Addison-Wesley, 2003.
- [11] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *WICSA*, 2001.
- [12] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *TSE*, 35(4), 2009.
- [13] The Apache Modeling Project. [www.fmc-modeling.org/projects/apache](http://www.fmc-modeling.org/projects/apache), 2008.
- [14] B. Gröne, A. Knöpfel, and R. Kugel. Architecture Recovery of Apache 1.3 – a Case Study. In *Intl. Conf. on Softw. Eng. Research and Practice*, 2002.
- [15] D. R. Harris, H. B. Reubenstein, and A. S. Yeh. Reverse Engineering to the Architectural Level. In *ICSE*, 1995.
- [16] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *TSE*, 27(2), 2001.
- [17] J. F. Kelley. An empirical methodology for writing user-friendly natural language computer applications. In *SIGCHI Conference on Human Factors in Computing Systems*, 1983.
- [18] B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4), 1995.
- [19] R. Koschke. Architecture Reconstruction: Tutorial on Reverse Engineering to the Architectural Level. In A. D. Lucia and F. Ferrucci, editors, *International Summer School on Software Engineering*, 2008.
- [20] R. L. Krikhaar. Reverse Architecting Approach for Complex Systems. In *ICSM*, 1997.
- [21] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *TSE*, 21(9), 1995.
- [22] MagicDraw. MagicDraw UML. [www.magicdraw.com](http://www.magicdraw.com).
- [23] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-To Analysis for Java. *TOSEM*, 14(1), 2005.
- [24] D. Perry and A. Wolf. Foundations for the Study of Software Architecture. *Softw. Eng. Notes*, 17(4), 1992.
- [25] M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In *POPL*, 1999.
- [26] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using Dependency Models to Manage Complex Software Architecture. In *OOPSLA*, 2005.
- [27] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. *TSE*, 32(7), 2006.
- [28] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *TSE*, 21(4), 1995.
- [29] M. Shaw and D. Garlan. *Software Architectures: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [30] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code*. Springer-Verlag, 2004.
- [31] R. T. Tvedt, P. Costa, and M. Lindvall. Does the Code Match the Design? A Process for Architecture Evaluation. In *ICSM*, 2002.
- [32] V. Tzerpos and R. C. Holt. A Hybrid Process for Recovering Software Architecture. In *CASCON*, 1996.
- [33] xADL. [www.isr.uci.edu/projects/xarchuci/](http://www.isr.uci.edu/projects/xarchuci/).