Static Extraction and Conformance Checking of the Runtime Architecture of Object-Oriented Systems

Marwan Abi-Antoun
School of Computer Science, Carnegie Mellon University
marwan.abi-antoun@cs.cmu.edu

Abstract

We propose a semi-automated approach to statically extract a runtime architecture of an object-oriented system using ownership domain annotations and check its structural conformance with an as-designed architecture. In contrast, previous work used dynamic analyses that cover some but not all possible executions, or used radical language extensions or implementation restrictions. The approach also extracts an architecture that is hierarchical and *sound*, i.e., one that accounts for all objects and relations that could possibly exist at runtime.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Experimentation, Languages

Keywords architectural extraction, conformance checking, ownership types, runtime architecture

1. Introduction

The field of software architecture has long recognized the importance of *runtime architectures* that model runtime entities and their potential interactions. To date, much of the available tool support has focused on the *code architecture*. In particular, extracting the runtime architecture of object-oriented systems remains hard.

Many approaches use a mix of dynamic and static information such as naming conventions and directory structures as well as runtime traces, and require the extractors to use trial and error with various clustering algorithms. Dynamic analyses often extract partial descriptions that cover the interactions between objects for the exercised use cases. An architecture is meant to capture a complete description of the system's runtime structure. For instance, an architectural-level security analysis needs a complete architectural description to handle the worst, instead of the typical case of runtime component communication. To meet this goal, a static analysis is needed. Moreover, the analysis must be *sound* and account for all objects and relations that could possibly exist at runtime.

The key insight of this work is to leverage ownership types for architectural extraction. Ownership types were originally proposed to control aliasing [4], but they are suitable for this problem because they track instances not classes, and the problem calls for an instance-based view. This insight leads to a principled approach to architectural recovery. A developer guides the architectural abstraction by adding annotations to the source code to clarify the architectural intent related to object encapsulation, logical containment (hierarchy), architectural tiers and object communication permissions. The idea of using annotations to recover a design from the code is not new [6]. But previous annotation-based systems did not specify the runtime instance structure or data sharing precisely or handle inheritance [6].

The approach does have the overhead of adding the annotations to a program, currently done mostly manually. Precise and scalable ownership inference is a separate problem and an active topic of ongoing research.

2. Contributions

This work provides three primary contributions to object-oriented development: (a) a more flexible ownership domains type system to annotate existing complex object-oriented code without having to refactor; (b) a static analysis to extract a sound hierarchical runtime architecture from an annotated program; and (c) a semi-automated analysis to check and measure the structural conformance of an as-built system to an as-designed runtime architecture.

Ownership Domains Type System. We extended the ownership domains type system by Aldrich and Chambers [4] to address expressiveness challenges we identified while annotating real code. In addition to the traditional goals of ownership types to detect encapsulation failures, we studied their relevance to help identify tight coupling, object borrowing issues and other design problems.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

Compared to using radical language extensions [9] or specific implementation substrates, these annotations support existing languages, designs, frameworks and libraries.

Architectural Extraction. In addition to the ownership annotations, the developer indicates which types are more architecturally relevant than others. This enables the analysis to abstract objects by ownership hierarchy and by types. Previous research has indicated that fully automated abstractions often do not match the architect's mental model [7].

Existing extraction of runtime architectures uses dynamic analyses or a mix of static and dynamic analyses. Dynamic analyses show the ownership structure in a few program runs. But a compile-time representation shows ownership relations that will be invariant over all program runs.

The approach soundly handles possible aliasing and inheritance. Compared to non-hierarchical object graphs obtained from a program without annotations [5, 8] or with other annotations [6], the extracted architecture is hierarchical. In our experience, hierarchy is indispensable to obtain an architecture at a meaningful abstraction level, and make it possible for a tool to compare it to what an architect might draw for an as-designed architecture.

Theory. From the abstract syntax tree of the annotated program, the analysis builds several intermediate representations. We formally specified the transformation steps using rewriting rules and proved key soundness theorems [2]. The soundness proof relates the runtime store to the extracted architecture and ensures that the analysis maps each object in the true runtime object graph to exactly one object in the architecture, and accounts for all relations between objects.

Architectural Conformance. A system conforms to its as-designed architecture if the latter is a conservative abstraction of the system's runtime structure. By the *communication integrity* principle, *each component in the implementation may only communicate directly with the components to which it is connected in the architecture*.

View synchronization assumes that both views are equally important and attempts to make them identical [3]. In contrast, conformance checking is asymmetric because the as-built view often contains details that are irrelevant to the as-designed view. Guided by the principle above, conformance checking accounts for any communication in the as-built view that is not in the as-designed view, without cluttering the as-designed view with implementation details [1].

The approach complements architectural conformance approaches that focus on the code architecture [7]. The approach relates source code entities to an as-designed hierarchical runtime architecture. It also compares the as-built and the as-designed architectures using a structural comparison that works with hierarchical architectural views, does not assume unique identifiers, detects renames and moves and allows the user to force or prevent matches [3]. These relaxed assumptions more closely match the problem of the post-hoc conformance checking of runtime architectures.

Evaluation. We implemented tool support for the approach and evaluated it on real object-oriented code to answer the following research questions: Does the extracted architecture have a meaningful level of abstraction? Can the conformance check identify interesting architectural structural non-conformities in existing systems?

Extended Examples. During the development and the refinement of the approach, we used several extended examples totaling 38 KLOC [2]. While these sizes seem small, the static analysis of runtime architectures is less mature than that of code architectures. For instance, the most relevant previous work was applied to one 1.7 KLOC system [6].

Field Study. We conducted a week-long on-site field study whereby we analyzed a 30 KLOC module from a proprietary commercial system of over 250 KLOC. In a few days, we were able to add the annotations to the module and extract a top-level architecture for review by a developer. We are currently refining the approach based on the experience.

3. Conclusion

This work brings the object-oriented development practice closer to statically extracting and checking the conformance of the runtime architecture of existing systems.

Acknowledgements. This is joint work with my Ph.D. advisor, Jonathan Aldrich. My thesis committee, Nenad Medvidovic, Brad Myers and William Scherlis also guided me.

References

[1] M. Abi-Antoun and J. Aldrich. Static Conformance Checking of Runtime Architectural Structure. CMU-ISR-08-132, 2008.

[2] M. Abi-Antoun and J. Aldrich. Static Extraction of Object-Oriented Runtime Architectures. CMU-ISR-08-127, 2008.

[3] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, and D. Garlan. Differencing and Merging of Architectural Views. *ASE*, 15(8):35–74, 2008.

[4] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.

[5] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *TSE*, 27(2), 2001.

[6] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, 2003.

[7] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE TSE*, 27(4), 2001.

[8] R. W. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, CMU, 2001.

[9] J. Schäfer, M. Reitz, J.-M. Gaillourdet, and A. Poetzsch-Heffter. Linking Programs to Architectures: An Object-Oriented Hierarchical Software Model based on Boxes. In *The Common Component Modeling Example: Comparing Software Component Models*, LNCS. Springer, 2008.