To appear in ICSE-96

Monitoring Compliance of a Software System With Its High-Level Design Models

Mohlalefi Sefika* Aamod Sane† Roy H. Campbell

University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue, Urbana, IL 61801
email: {*sefika,sane,roy*}*@cs.uiuc.edu*
www: `http://choices.cs.uiuc.edu`

Abstract

As a complex software system evolves, its implementation tends to diverge from the intended or documented design models. Such undesirable deviation makes the system hard to understand, modify, and maintain. This paper presents a hybrid computer-assisted approach for confirming that the implementation of a system maintains its expected design models and rules. Our approach closely integrates logic-based static analysis and dynamic visualization, providing multiple code views and perspectives. We show that the hybrid technique helps determine design-implementation congruence at various levels of abstraction: concrete rules like coding guidelines, architectural models like design patterns[7] or connectors[26], and subjective design principles like low coupling and high cohesion. The utility of our approach has been demonstrated in the development of μ*Choices*, a new multimedia operating system which inherits many design decisions and guidelines learned from experience in the construction and maintenance of its predecessor, *Choices*.

1 Introduction

The construction of complex software systems is often guided by high-level design models and guidelines that improve the system quality by enabling experience reuse. Such guidelines are diverse in nature, ranging from environment-specific programming conventions, to rules-of-thumb like the information hiding principle, to domain-independent design idioms. Recently, research in software architecture has focused on identifying and cataloging reusable design models and rules, including object-oriented patterns[7], architectural styles[8], and software connectors[26].

Unfortunately, system design improvements are often vitiated by the tendency of implementations to diverge from their intended or documented design models. This suggests that the use of codified design principles must be supplemented by checks to ensure that the actual implementation adheres to its design constraints and guidelines. Without easy and effective verification, the deviation of the implementation from its design commitments hurts even systems that arduously attempt to reuse proven experiential knowledge, making them hard to understand, modify, and maintain.

This paper presents a hybrid computer-aided approach that enables close monitoring of the implementation's faithfulness to its posited design abstractions at all stages of system development. The verification process starts as early as possible with logic-based static analysis of source code, complemented systematically with dynamic interactive visualization for the inspection of run-time behavior. Such a multi-dimensional approach reveals aspects of system implementations that escape either static or dynamic analysis alone. Frequent compliance checking helps catch design turnovers before they become irreversible.

Surprisingly, there seem to be few preventive tools whose specific purpose is to help assure that the implementation of an evolving system abides by its imposed design principles. Those that we know (e.g., [16, 15, 13]) appear to focus exclusively on either static or dynamic analysis, lacking multiple views and perspectives. We improve on these approaches in several ways:

- We integrate static analysis and dynamic visualization. We show that the combined scheme improves compliance checking for well-defined rules as well as subjective guidelines.
- We check for conformance to a wide variety of design principles: *concrete* rules like coding styles[14], *ar-*

* Supported by Lesotho AFGRAAD Fellowship.
† Supported by CNRI contract CNRI GIGABIT/UILL.

chitectural rules like design patterns[7] or styles[8], and *heuristic* guidelines like high cohesion and low coupling[20].

- We provide a design rule-base that is *reusable* across a wide variety of software systems, because the rule-base captures codified architectural styles, connectors, and design patterns that are well-known recurrent designs.

Our compliance checker, called *Pattern-Lint*[1], uses models from various works[7, 8, 26, 14, 6], and our experience in designing *Choices*[2], one of the earliest object-oriented operating systems. We are using *Pattern-Lint* to implement μ*Choices*[3], a successor of *Choices* targeted to support multimedia applications. *Pattern-Lint* helps reuse and refine structures from *Choices* in μ*Choices*. The tool aids in monitoring implementation compliance from the very beginning. We do not want the new operating system to diverge from its intended design abstractions during development, and then face painful reverse-engineering of "rotten" structures during maintenance. We have learned through bitter experience that prevention is better than cure.

Overview We begin the paper with a series of examples that demonstrate *Pattern-Lint*'s effectiveness. Section 2 illustrates consistency checking for architectural rules like design patterns, and the role of static and dynamic visualization in detecting discrepancies. Section 3 shows how highly parameterized static and dynamic visualizations prove critical for analyzing very abstract and subjective design properties like coupling and cohesion. Section 4 gives examples of more concrete rules, further illustrating the flexibility and generality of our approach. Section 5 presents our scheme in detail and examines its limitations. Section 6 reports on related work. Section 7 proposes future directions and concludes the paper.

2 Example: Static Abstractions and Dynamic Visualization

In this section, we discuss how *Pattern-Lint* was used to re-design a portion of the device management framework in *Choices*[11]. We briefly discuss the pattern underlying the framework's design. Then we show why conformance of the framework to the pattern is vital, followed by examples of the use of *Pattern-Lint* in the hybrid conformance checking process.

[1]Just as *lint* checks for bad coding practice and likely errors in C programs, *Pattern-Lint* checks for design violations and flaws.

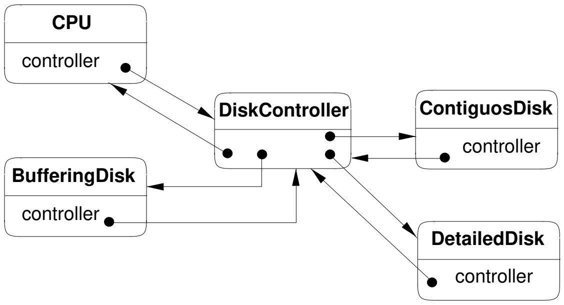

Figure 1: Device management framework (excerpt).

The Design: A Mediator Figure 1 depicts in OMT[22] notation a fragment of the device management framework in *Choices*. The framework allows CPU's and Devices to interact. In the figure, the classes CPU and Disk model the corresponding hardware resources. The Disk class has several subclasses, of which we show BufferingDisk, ContiguousDisk, and DetailedDisk. Instances of Disk and CPU need to interact, since CPU's must request disks to do I/O and the disks must in turn interrupt CPU's upon I/O completion. Our framework abstracts the interaction protocol in a DiskController, which centralizes the protocol, and simplifies coordination between multiple CPU's and disks.

The arrangement of Figure 1, in which a central component provides a localized home for the interaction semantics of several colleagues, is the structure of the MEDIATOR pattern[7]. The pattern reifies and encapsulates the collaboration protocol among several objects into a central hub: the mediator object[2]. This centralizes and objectifies the interaction protocol which would otherwise be scattered and distributed across collaborating objects, and hence difficult to locate, understand, and change. The pattern also decouples the collaborating colleagues and manages dependencies. Further discussion can be found in [7].

Conformance: Mediator Implementation The reason for our choice of the MEDIATOR design model in device management is clear. By centralizing the complex interactions between Disk and CPU in the DiskController, we make it easier to understand the code, analyze it about possible races or deadlocks, or alter it to add different devices. Now, suppose the implementation did not actually conform to the documented MEDIATOR pattern, because of a design turnover and documentation drift. In that case, some of the supposedly mediated colleagues interact directly, thereby adding unexpected interactions not visible by examining DiskController. Such obscure interactions would invalidate any correctness analyses based on examining the me-

[2]A mediator object is in effect a first class connector[26]

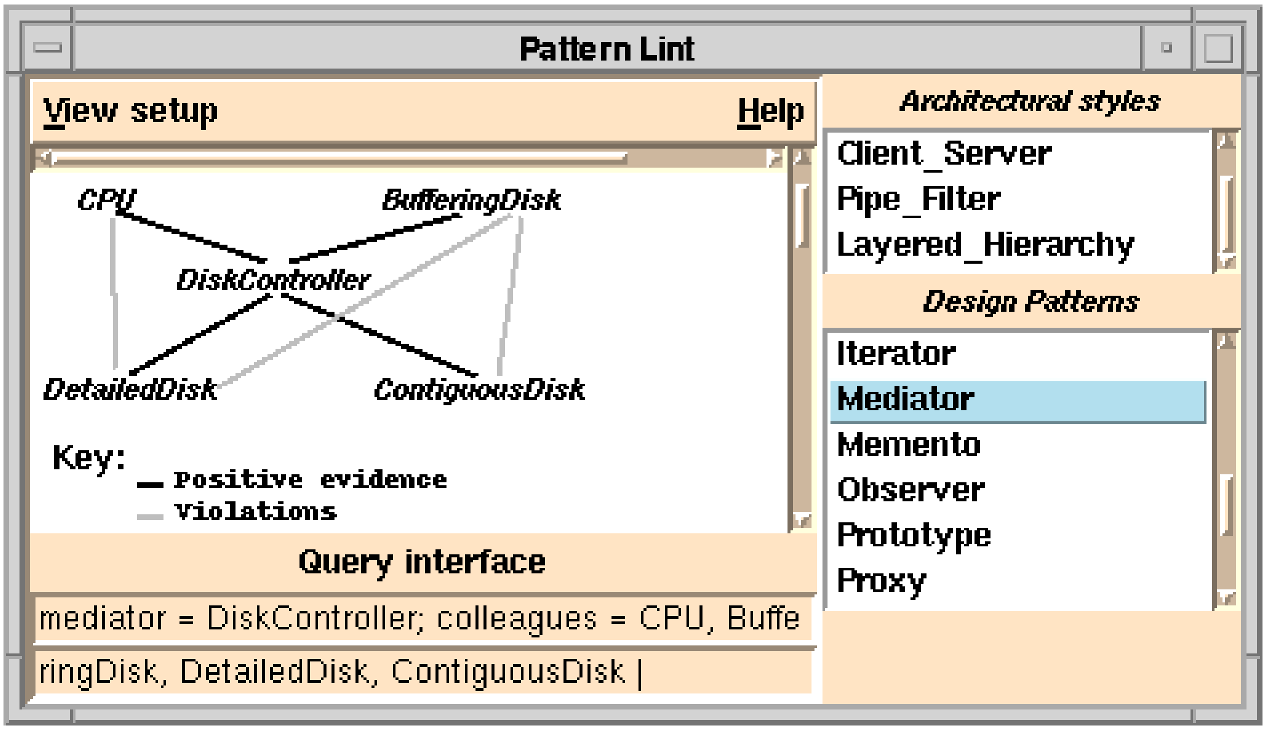

Figure 2: Comparison of the device management framework and MEDIATOR. The user simply selects the MEDIATOR design model from the browser and queries the expected class roles. Rules for conformance checking come from a pattern recognition knowledge base that is reused and shared by all users.

diator alone. This creates a potential for dangerous and risky software engineering decisions when a different programmer mistakenly believes that the implementation is a mediator. Thus, it is important that the implementation is actually a mediator.

Static analysis: Violations of Mediator When we imported the device framework from *Choices* into μ*Choices*, we applied *Pattern-Lint* to the subsystem. The findings from static analysis (see Figure 2) seemed to indicate that the implementation was inconsistent with MEDIATOR. It appeared that some programmers, in an attempt to improve performance, allowed CPU's to access directly certain device operations. Specifically, the C++ `friend` declaration was used to circumvent normal C++ protection to grant direct access. In addition, some disk devices were also unexpectedly bypassing their mediator and communicating directly with the CPU.

Figure 2 tells us that there is some evidence that the structure may be a MEDIATOR — the DiskController, Disks and CPU interact as required (*positive evidence*). However, the diagram also reports that some mediated instances ignore the mediator and communicate directly through the friendship mechanism, indicating definite *violations*[3] of the constraints imposed by the mediator design model. All of these relationships are determined directly from the code using a Prolog database and inference engine (see Section 5).

Visualization: A Surprise Next, guided by the results from static analysis, we instrumented the framework mechanically through a graphical instrumentation interface. The findings and suspicions gathered from static analysis served as a requirements specification for the run-time information to visualize. To graphically analyze system dynamics, *Pattern-Lint* relies on *OS View*[25], the visualization and manipulation system for *Choices*.

Figure 3 is a snapshot of an animation showing the dynamic interactions among the target classes. In the diagram, the thickness of the line connecting two classes corresponds to their relative interaction frequency. This animation was fairly consistent across many application workloads.

The animation reveals that the DiskController interacts heavily with DetailedDisk, less heavily with CPU and ContiguousDisk, and not at all with BufferingDisk. More importantly, the animation shows *no* significant direct interaction between the mediated colleagues that shows at the

[3] The grey edges are hyperlinked to the associated source code.

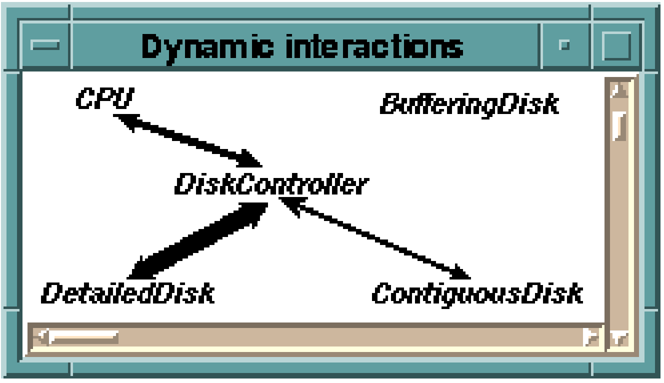

Figure 3: Animating method calls of the device management framework.

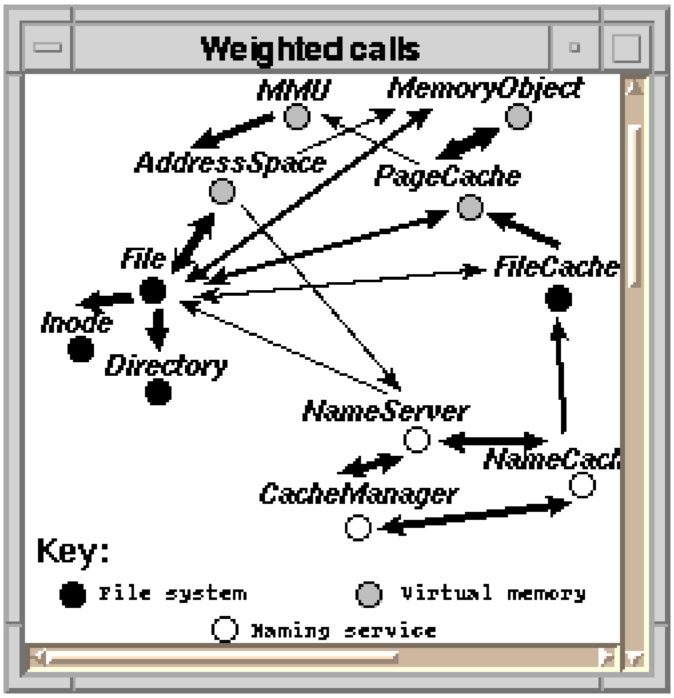

Figure 4: Static weighted inter-class call graph.

scale of the diagram. Thus, visual analysis brings to light results that were not at all discernible from static analysis:

- BufferingDisks are not used in this particular instantiation of the system; their initialization was a waste of system resources.
- From the performance point of view, the structure *is* a MEDIATOR. It turns out that the performance hacks leading to static violations of MEDIATOR were both unnecessary and unfruitful.

Thus, correlating the static and dynamic code views, we detected potentially dangerous inconsistencies, and also discovered how to get rid of them.

3 Example: Heuristic analysis of subsystem cohesion and coupling

The previous experiment checks conformance to an explicit and well-defined architectural model. In contrast, this experiment demonstrates compliance verification for a highly abstract and subjective design principle: high cohesion and low coupling. Lacking a standard measure of cohesion and coupling, the design principle is best treated heuristically, rather than by a set of formal rules[24].

Motivation We are interested in application-specific customization of subsystems in μ*Choices*. Thus, it is desirable that the subsystems be as self-contained and independent of one another as possible, so that modifying a subsystem has minimal unanticipated side-effects on its neighbors. In the abstract, this requirement is easier to fulfill in systems with low coupling and high cohesion[20].

Static analysis: method calls Figure 4 displays a weighted inter-class call graph within and across three *Choices* subsystems: *virtual memory*, *file* system, and *naming* service. The graph heuristically clusters and interconnects classes according to the number of static interclass message invocations counted from the source code, giving messages that appear in control loops more weight. Ideally, a cluster should correspond to a functionally coherent subsystem. From the figure, we see that the *naming* subsystem appears to be fairly self-contained and decoupled. On the other hand, the *virtual memory* and the *file* systems are not so well isolated. For example, the FileCache class seems to interact more heavily with classes in other subsystems than in its own subsystem. Modifying this class to customize the file system could potentially propagate unexpected bugs and performance flaws throughout the other subsystems.

Static analysis: data sharing A different perspective of coupling, based on the degree of data sharing, is shown in Figure 5. The figure clusters and interconnects classes depending on the number of shared variables and their usage. According to both the call graph and the data sharing graph, the *naming* system is the most well isolated of the subsystems. But the data sharing graph suggests that the other systems are also fairly independent as regards information sharing. For example, the suspect FileCache class

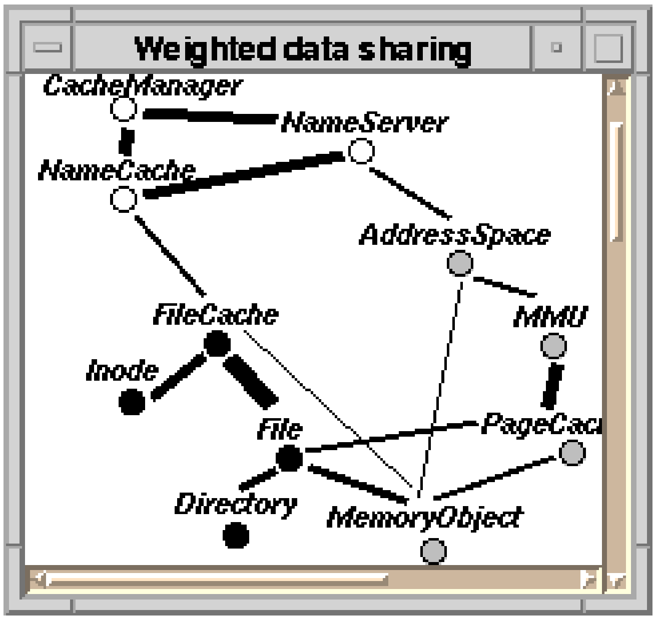

Figure 5: A weighted data sharing graph.

belongs more to its own subsystem.

Comparing the two graphs, we conclude that excess coupling is likely introduced through method calls. The subsystems are probably cohesive with respect to information sharing, dependency, and management.

Yet another perspective: dynamic visualization A dynamic version of the weighted call graph of Figure 4 is shown in Figure 6. Initially, the target classes are placed arbitrarily on the display. During the animation, classes attract each other if they communicate frequently, otherwise they repel. The scalable affinity diagram[4] reveals the *runtime* communication structure of the *Choices* subsystems, and is fairly consistent over different workloads.

Comparing the static and dynamic call graphs, we gain a high degree of confidence that the actual subsystem behavior exhibits high cohesion and low coupling. For instance, the suspicious collaborations from the static diagram (e.g., between FileCache and PageCache, and between File and PageCache) do not appear to be seriously exercised – they are not manifest on the scale of the diagram. Indeed, the explicit subsystem clusters in the animation correspond directly to the documented subframeworks of *Choices*[2].

Putting it all together Correlating the static and dynamic call diagrams indicates that the undesirable couplings from

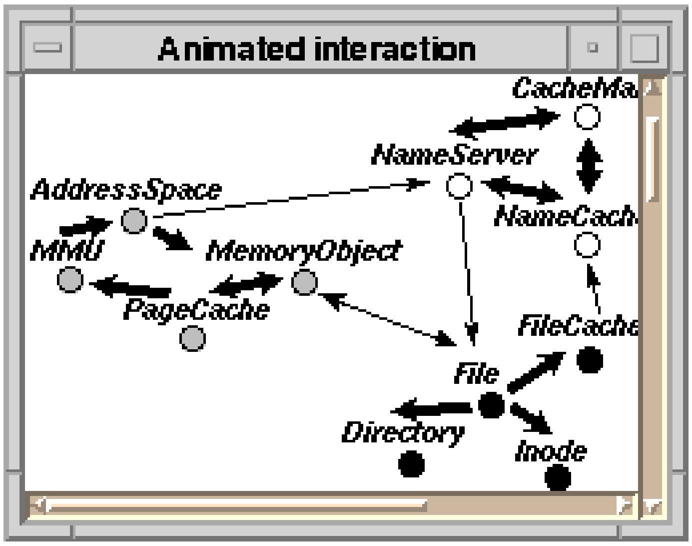

Figure 6: An affinity diagram animating dynamic interclass call relations.

the static diagram presumably model unusual or uncommon situations, and that they might unnecessarily entangle the subsystems. The data sharing graph reveals that the unwanted connections may have been introduced as method calls.

Upon examining the source code, we discovered that FileCache and PageCache were originally the same component. When the component was split apart during restructuring, unintended residual dependencies from FileCache remained in PageCache. This not only introduced obscure interactions between FileCache and PageCache, but also tightly coupled File with PageCache. Moving the dependencies to FileCache made the three components functionally cohesive, and also clarified their roles in the subsystems.

Discussion The preceding example shows the importance of multiple perspectives in checking designs for heuristic properties like coupling and cohesion. In particular, using static analysis of calls, we discovered that the sub-systems were not as cleanly segregated as expected. Changing the design parameter to data sharing helped realize that this was not due to data sharing. Since the static analysis results indicated that subsystem couplings were likely due to method calls, we put more emphasis on dynamic visual analysis of inter-object calls, not on inter-object data access. Visual analysis of suspicious calls indicated that some uncommon cases were overlooked in splitting the original PageCache. All these findings guided our efforts to better configure and organize the components and subsystems, reducing unwanted coupling, increasing cohesion,

[4] The affinity diagram is scalable because its format, meaning, clarity, and size are independent of the number of dynamic instances of the classes involved[5].

and eliminating unnecessary compilation dependencies and costs.

4 Example: Miscellaneous Guidelines

The previous examples have shown the utility of *Pattern-Lint* in evaluating complex design models. In this section, we describe compliance checking for simple, concrete guidelines. In the following, the precise syntactic checks made by *Pattern-Lint* are italicized.

Avoid Inheritance/Reference Cycles: A common structure often seen in object-oriented systems is an inheritance hierarchy rooted at class Object. This root class supports basic behavior for I/O or debugging calls by delegating them to member objects that actually implement the operations. But the I/O and debugging member objects themselves are derived from Object. Hence, behavior of a *base* class becomes dependent on the behavior of a *derived* class, and modifying a derived class may indirectly affect other children of the base class. Except for special situations (e.g., if it is known that the base and derived classes will not be changed), Inheritance/Reference cycles that introduce hidden associations are highly undesirable.

In *Choices*, when such cycles were accidentally established across subsystems, the resulting interdependencies seriously impaired portability and comprehensibility[23]. In μ*Choices*, we are using *Pattern-Lint* to ensure that *no base class holds references to instances of its derived classes*.

Substitutability via refinement by addition: Frameworks[6] are groups of collaborating classes that typically implement subsystems in object-oriented programs. For example, a PageCache and its associated PagingPolicy form a simple framework that implements demand paging in *Choices*. To add new capabilities to the system, frameworks are *refined* to derive other frameworks.

Ideally, the *derived* framework should remain substitutable in place of the *original* one, so that the rest of the system is not impacted negatively by framework refinement. A simple way to promote substitutability is to always refine frameworks only by *adding* new components, and never *removing* any existing component[2]. While component addition does not guarantee substitutability (which is a complex behavioral notion), component removal almost always violates it.

In our example, suppose that the default PagingPolicy implements FIFO (first-in/first-out) paging. One may legally refine this framework by adding a LRU (least recently used) paging policy class. *Pattern-Lint* verifies that *given a set of components that define a framework, no derived framework has fewer components in it*.

No Inheritance across subsystems: Subsystems interact by allowing access to their component classes. In our work with *Choices*, we observed that if subsystems export classes that contain data members or define subsystem behavior, developers are tempted to subclass these classes across the subsystems in order to access the data or alter the behavior. This exposes the internals of one subsystem to another, increases the likelihood of unexpected side effects, and makes the subsystems very difficult to modify or understand independently.

To avoid this problem in μ*Choices*, *Pattern-Lint* confirms that *subsystems only communicate using abstract classes* that do not define data members or methods.

5 *Pattern-Lint* design and rationale

The design of *Pattern-Lint* addresses two main issues:

- How to express design properties to check compliance with design models at varying levels of abstraction.
- How to communicate system information in a highly selective and problem-specific manner.

5.1 Expressing Design Models

Pattern-Lint supports conformance checking at various levels of abstraction with multiple perspectives.

Low Level Rules Low level rules (the guidelines of Section 4) govern details closer to implementation than design. These rules are specified using ER diagrams or grammars and mechanically converted to their Prolog equivalents.

Architectural Rules These rules include patterns[7], software interconnection models[26], and architectural styles[8]. The rules capture recurring designs, so they are described only *once*, and reused across a wide variety of software systems.

We define architectural models formally using two types of Prolog clauses: (1) those that provide incremental confidence that the implementation may be faithful to the model, and (2) those that detect definite violations of the constraints imposed by the model. Combining and correlating these two types of knowledge helps reduce the risk of both *false positives* and *false negatives*.

Figure 8 shows an excerpt of the code in our predefined test driver for MEDIATOR. This pattern is open to slight implementation variations[7], so we need several sets of

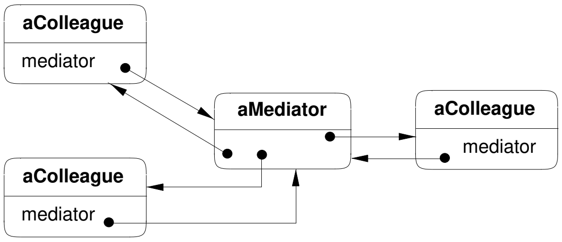

Figure 7: A MEDIATOR structure.

rules to cater for each expected variation. The rules in Figure 8 are appropriate for the mediator realization of Figure 7 expressed in OMT[22] notation. In the test driver, the positive evidence rule checks that (1) each colleague does interact with the mediator, and (2) there are indeed bidirectional control and data dependencies among the mediated colleagues. Success of the positive evidence rule alone does not prove compliance with MEDIATOR, as there may be constraint violations. So the violations rules look for definite inconsistencies with the pattern as exemplified by direct interaction among colleagues that bypasses their mediator, or failure of the mediator to communicate with a supposedly mediated colleague.

Heuristic Rules These rules cover highly abstract and subjective design guidelines such as cohesion and coupling or the information hiding principle. Since the rules are very general, they are difficult to specify formally as Prolog clauses. To check them, *Pattern-Lint* allows a designer to generate multiple relational views of a system like static call graphs and animated object interactions, and apply his/her own heuristics. The analysis of subsystem cohesion and coupling in Section 3 illustrates heuristic inspection.

5.2 Problem Oriented Visualization

Pattern-Lint implements design compliance checking primarily using static and dynamic visualization with multiple perspectives. The tool automates many aspects of the visualization process, supporting easy selection of the data of interest and smooth coordination of program constructs and multiple graphical views. Towards these goals, the tool uses the design models as the primary units for program visualization; they generate the necessary abstractions and also filter the program data. In particular, *Pattern-Lint* supports:

- *Model Driven Visualization:* Graphical views are generated and coordinated using the same Prolog clauses that define a model. The views are parameterized by the data of interest. For instance, the edges in a MEDIATOR display graph may represent or animate one or more of the relations *invokes*, *control-flow*, and *data-dependent*, according to the positive evidence rule in Figure 8. The viewer can zoom in and out of the graph and specify the relations to be shown or animated.
- *Automated and Coordinated Instrumentation:* Graphs generated by static analysis (e.g., Figure 2) are hyperlinked to the corresponding source code. By selecting edges or nodes, the user can automatically instrument the code in several ways.

  In addition, at runtime, a user can directly manipulate instruments[25]. For instance, the viewer can turn selected instruments on or off to improve the clarity of an affinity diagram like Figure 6. Similarly, the rate of data production (and hence system perturbation) can interactively be controlled. The instruments interoperate to generate coordinated views and select runtime information.

5.3 *Pattern-Lint* Architecture

Figure 9 depicts the architecture of *Pattern-Lint*. Reading from left-to-right, the diagram shows how information about program entities and their expected high-level model is processed to generate the static and dynamic program views.

The designer specifies program entities to be examined and the design model (see Figure 2). A parser and database generator analyzes the program source and creates a database of implementation-level relations for reasoning about the model. A Prolog inference engine matches the program data to the model rules, and produces relations that demonstrate consistency with or divergence from the design model.

A static correlator then maps these relations to the source code and generates browsable views. At this point, a designer may instruct an automatic instrumentor to instrument the code for selected relations. The instrumentor interface also enables user manipulation of instruments. Finally, a renderer maps the data and displays the views.

5.4 Limitations of our approach

Pattern-Lint uses three types of design models: *concrete*, *architectural*, and *heuristic*. Each model has limitations[5] in terms of verifiability, descriptive ability, and usability:

- *Concrete* models such as coding guidelines or control-flow diagrams are easy for compliance checking, but often contain operational detail that obscures design intentions.

[5] In addition to the well-known deficiencies of static analysis (e.g., pointer analysis) and dynamic analysis (e.g., input data sensitivity).

`/*` **Positive evidence:** `*/`
`  mediator(M1, C1 , C2 ) :-`
`      invokes(C1 , M1),`
`      invokes(M1 , C1),`
`      invokes(C2 , M1),`
`      invokes(M1 , C2),`
`      control_flow( C2 , C1 ),`
`      control_flow( C1 , C2 ),`
`      data_dependent( C1 , C2 ),`
`      data_dependent( C2 , C1 ).`
	`/*` **Violations:** `*/`
	`  violations_mediator(M1, C1 , C2 ) :-`
	`      invokes(C1,C2).`
	`  violations_mediator(M1, C1 , C2 ) :-`
	`      not(invokes(M1,C1)).`
	`  violations_mediator(M1, C1 , C2 ) :-`
	`      is_friend_of(C1,C2).`

Figure 8: A partial Prolog test driver for MEDIATOR.

- *Architectural* models like patterns are abstract, but have many possible implementations. The rule-base of *Pattern-Lint* captures many of the possibilities, but conformance checks cannot be absolutely guaranteed. We can cope with this limitation where feasible by expanding and refining our reusable knowledge base as we gain more experience.
- *Heuristic* models can be checked using dynamic and static visualization with surprising effectiveness. Nonetheless, the analysis for heuristic guidelines is rather human-dependent, and the certainty of compliance is much less.

6 Related work

Other systems for monitoring an implementation's faithfulness to its design models appear to have focused on either static or dynamic verification, but not both[16, 15, 24, 13]. We improve on these systems by systematic provision of numerous static and dynamic visual perspectives, and by considering a wide range of design commitments, from concrete implementation-level rules to configuration-level principles like subsystem cohesion and coupling. In particular, our work also concerns the conformance checking for standardized architectural models[26, 7, 8]: recurring design abstractions that are increasingly popular in the construction of complex systems.

Most of the individual features of *Pattern-Lint* have appeared in isolated reverse engineering tools: architectural styles[10], dynamic analysis[17, 21], program databases[4], and program concept recognition[1, 9]. *Pattern-Lint* seems to be the first tool to combine all these abilities in design conformance checking. *Pattern-Lint* also extends the application of these ideas by incorporating diverse design principles, celebrating the synergy of static and dynamic visualization, the use of scalable affinity diagrams, and a customizable architecture.

Another approach for improving design-implementation congruence provides improved notations to represent designs[22]. In limited domains such as interface generators[18], low-level implementations can automatically be generated, assuring design conformance. In using architectural catalogs, *Pattern-Lint* supports this move toward improved notations.

Our animated class interaction views were influenced by De Pauw et al.[19]. We extend their techniques with affinity diagrams that not only reveal clusters of frequently communicating classes, but also highlight key elements of the clusters and expose inter/intra-cluster dependencies. Finally, Lange and Nakamura [12] use patterns as a guide to manually select objects for visualization. In contrast, we have formalized the notion of patterns based on positive evidence and violations (expressed here as Prolog rules). This allows us to automate static and dynamic visualization and to check for design conformance.

7 Conclusion and Future Work

Once a software system deviates from its intended or documented design models, it becomes difficult and costly to understand, modify, and maintain. There have been two standard approaches to address this problem: reverse engineering[10] the implementation to repair it typically after the fact, and structuring the design using standardized design constructs[7, 8] with fairly well-understood corresponding implementations. Neither alternative alone seems to be satisfactory: reverse engineering researchers strongly point out the need for more careful forward engineering, and the software architecture literature is rife with warnings not to regard improved models as a panacea.

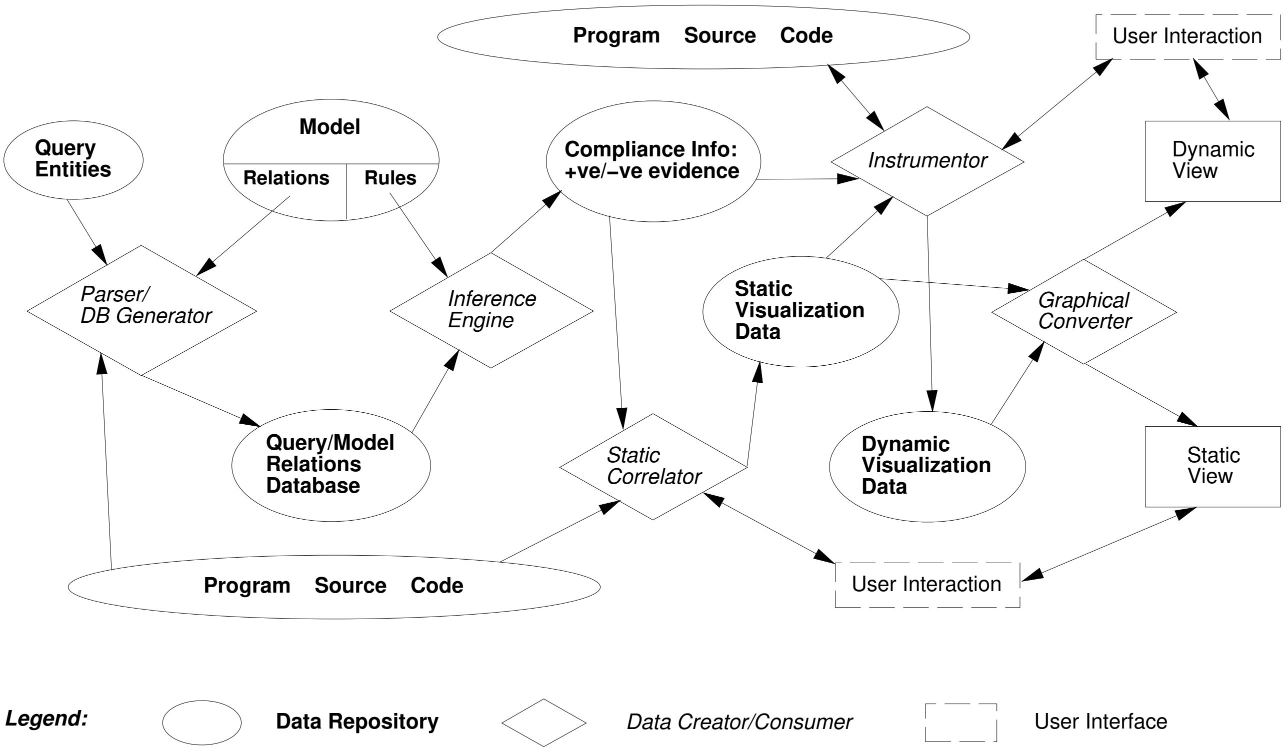

Figure 9: The Architecture of *Pattern-Lint*.

In this paper, we have shown that a combined approach that uses codified design models with "reverse engineering" style analysis of the implementation, applied very early in the system's life-cycle, is an effective forward engineering method. System development based on frequent compliance checking helps catch design turnovers before they corrode the implementation and become irreversible.

Our research suggests that systematic integration of logic-based static analysis with dynamic visualization, guided by the use of codified design knowledge, is highly effective in checking design conformance. First, we found that the same high-level models that structure the design also function as excellent primary units of conformance checking. Second, since design models appear at many levels of abstraction, it becomes important to employ multiple perspectives combining static and dynamic program information. Last, the models can be applied yet again as filters for the information that guides the selection, presentation, and analysis of data. The examples in the paper demonstrate how model-directed systematic correlation of static and dynamic data allows the investigation of intricate design properties and gives insight into new design alternatives.

One possible improvement to our approach involves incorporating generative capabilities into the checker. Derivation of code fragments and skeletons from design specifications can assure that certain aspects of the implementation conform to the design by construction. For example, *Pattern-Lint* could automatically generate the interclass references in a MEDIATOR implementation to guarantee a star-shaped network of collaboration. Other feasible extensions include run-time enforcement of implementation compliance through assertions annotated in code. We expect that model-driven dynamic "debugging" of design properties will prove useful.

Acknowledgments: We thank Kent Beck, Gail Murphy, Ellard Roush, and the ICSE referees for their helpful comments.

References

[1] T. J. Biggestaff, B. G. Mitbander, and D. Webster. The Concept Assignment Problem in Program Understanding. In *Proceedings of Working Conference on Reverse Engineering*, Baltimore, MD, USA, 1993.

[2] R. H. Campbell, N. Islam, D. Raila, and P. Madany. Designing and Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM*, pages 117–126, September 1993.

[3] R. H. Campbell and S. Tan. μChoices: An Object-Oriented Multimedia Operating System. In *Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, Washington, May 1995. IEEE Computer Society.

[4] Y. F. Chen, M. Y. Nashimoto, and C. V. Ramamoorthy. The C Information Abstractor System. In *IEEE Transactions on Software Engineering, Vol. 16, NO. 3,*, March 1990.

[5] A. L. Couch. Categories and Context in Scalable Execution Visualization. *The Journal of Parallel and Distributed Computing 18*, pages 195–204, December 1993.

[6] L. P. Deutsch. Design Reuse and Frameworks in the Smalltalk-80 Programming System. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume II, pages 55–71. ACM Press, 1989.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[8] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of the 2nd ACM SIGSOFT*, pages 175–188, December 1994.

[9] M. Harandi and J. Q. Ning. Knowledge-based Program Analysis. *IEEE Software*, January 1990.

[10] D. R. Harris, H. B. Reubenstein, and A. S. Yeh. Reverse Engineering to the Architectural Level. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, USA, April 1995.

[11] Panos Kougiouris. A Device Management Framework for an Object-oriented Operating System. Technical report, The University of Illinois at Urbana-Champaign, May 1991.

[12] D. Lange and Y. Nakamura. Interactive Visualization of Design Patterns Can Help in Framework Understanding. In *OOPSLA*, pages 342–357, 1995.

[13] D. Luckham, F. von Henke, B. Krieg-Bruckner, and O. Owe. *Anna, A Language for Annotating Ada Programs: Reference Manual, vol. 260 of Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[14] S. Meyers. *Effective C++: 50 Ways to improve Your Programs and Designs*. Addison-Wesley, 1992.

[15] S. Meyers, C. K. Duby, and S. P. Reiss. Constraining the Structure and Style of Object-Oriented Programs. Technical Report CS-93-12, Brown University, 1993.

[16] G. C. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In *Proceedings of the Third ACM Symposium on the Foundations of Software Engineering*, 1995.

[17] D. P. Olshefski and A. Code. A Prototype System For Static and Dynamic Program Understanding. In *Proceedings of the Working Conference in Reverse Engineering*, Baltimore, MD, USA, 1993.

[18] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Massachusetts, 1994.

[19] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the Behavior of Object-Oriented Systems. In *OOPSLA*, pages 326–337, October 1993.

[20] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc, 1992.

[21] Herbert Ritsch and Harry M. Sneed. Reverse Engineering Via Dynamic Program Analysis. In *Proceedings of Working Conference on Reverse Engineering*, Los alamitos, CA, USA, 1993.

[22] J. Rumbaugh, M. Blaha, W. Premerlani, and F. Eddy. *Object-Oriented Modeling and Design*. Prentice Hall, Eaglewood Cliffs, NJ, 1991.

[23] A. Sane and R. H. Campbell. Detachable Inspector: A Structural Pattern for Designing Transparently Layered Services. In *Pattern Languages of Programs (To appear)*, 1995.

[24] R. Schwanke. An Intelligent Tool for Reengineering Software Modularity. In *Proceedings of the 13th International Conference on Software Engineering*, May 1991.

[25] M. Sefika and R. H. Campbell. An Open Visual Model For Object-Oriented Operating Systems. In *Fourth International Workshop on Object Orientation in Operating Systems*, Lund, Sweden, August 1995.

[26] M. Shaw, R. DeLine, and G. Zelensnik. Abstractions and Implementations for Architectural Connections. Technical Report CMU-CS-95-136, CMU, March 1995.