Architectural Pattern Definition for Semantically Rich Modular Architectures

Joeri Peters, Jan Martijn E.M. van der Werf, Jurriaan Hage
Department of Information and Computing Sciences, Utrecht University, the Netherlands
{J.G.T.Peters, J.M.E.M.vanderWerf, J.Hage}@uu.nl

***Abstract*—Architectural patterns represent reusable design of software architecture at a high level of abstraction. They can be used to structure new applications and to recover the modular structure of existing systems. Techniques like Architecture Compliance Checking (ACC) focus on testing whether the realised artefacts adhere to the architecture. Typically, these techniques require a complete architecture as input. In this paper, we focus on defining architectural patterns in such a way that we can use ACC tools to recognise architectural pattern instances. This requires us to explicitly define architectural patterns in terms of allowed and disallowed software dependencies. We base ourselves on Semantically Rich Modular Architectures. Defining architectural patterns this way allows us to reason about them. For example, how patterns should be interpreted as incomplete architectures and how different interpretations affect the pattern recognition process. Recognising architectural patterns using ACC techniques also has great potential in architecture design and Software Architecture Reconstruction.**

I. Introduction

"The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both" [1]. One can distinguish three types of structures within software architecture (static, dynamic and allocation) [1], or look at a categorisation based on views and viewpoints, such as functional, development, and concurrency [2]. This paper is restricted to static, technical architectures, the branch of software architecture that looks at the arrangement of source code into modules such as components and layers (i.e. the module view). We study their connecting dependencies without incorporating runtime behaviour.

Architectural patterns prevent software architects from reinventing the wheel in much the same way as design patterns do for developers. They establish the relationship between a context, a problem and a solution [1]. Examples of such patterns include Model-View-Controller, Broker and N-Layered patterns. Another concept related to architectural patterns is architectural *style* [3]. We recognise the two terms are often used interchangeably, but we prefer saying that the layering of a system is its style, whereas the layering within, for example, one of a system's modules is an instance of the pattern. This is merely a distinction in scope, allowing us to state that a system has an MVC style with a 3-Layered pattern in the Model-module, for example.

This paper describes our ongoing research into pattern recognition using Architecture Compliance Checking (ACC). We want to use ACC tools to assist the challenging process of Software Architecture Reconstruction (SAR) by searching for hypothesised parts of the architecture, e.g. architectural patterns, thereby deriving the intended architecture. ACC works by checking whether realised artefacts, such as source code, adhere to the intended architecture [4]. This can be to check the developers correctly implemented the architecture, or that the documentation has been kept up to date. Static ACC is thus aimed at the analysis of software dependencies between static modules and their correspondence to architecture documentation. This is but a portion of a larger reconstruction approach, since there are limitations to how much of the architecture can be derived from code. A complete SAR method would have to go beyond just the module view and be able to combine different code bases. ACC relies on a proper and adequate architecture description, which is typically tool dependent. We are thus faced with the problem of defining architectural patterns using such descriptions, which this paper addresses.

"A semantically rich modular architecture includes modules of semantically different types, while a variety of rule types may constrain the modules" [5]. Design and validation of such Semantically Rich Modular Architectures (SRMAs) is possible with the ACC tool HUSACCT, which distinguishes itself by supporting elements commonly used in SRMAs [6]. These elements can be said to be HUSACCT's architectural language. Several tools (e.g. SAVE, Lattix and Sonargraph) have previously been compared by the creator(s) of HUSACCT [7] and this open-source tool has been made to have extensive and configurable SRMA support, making it interesting for academic purposes.

HUSACCT is intended to describe a complete, as opposed to partial, architecture. As we are interested in the potential role of ACC to check whether an architectural pattern is present in source code, we want to study the possibilities this tool's language provides to express architectural patterns in terms of allowed and disallowed dependencies. HUSACCT does distinguish between dependency types, such as method calls and variable accesses, but this is not relevant to our pattern definitions at this time. We thus focus on the following research question: "how can architectural patterns be expressed in HUSACCT's terms and what are the consequences of alternative pattern definitions?"

The remainder of this paper is structured as follows: Section 2 explains the relevant portions of HUSACCT's SRMA support, in Section 3 the consequences of pattern definitions for the allowed dependencies are investigated, Section 4 shows

978-1-5090-2131-4/16 $31.00 © 2016 IEEE
DOI 10.1109/WICSA.2016.50

TABLE I
THE 7 RELATION RULE TYPES, SPLIT INTO TWO SUB-CATEGORIES.

Is not allowed to use
Is not allowed to back-call (layers)
Is not allowed to skip-call (layers)
Is allowed to use
Is only allowed to use
Is the only module allowed to use
Must use

how these choices become critical when combining patterns, Section 5 lists some of the relevant literature and Section 6 portrays the possible future industrial impact of our work.

II. SRMA CONCEPTS

SRMAs combine various types of modules with rule types to express architectural elements and their constraints. This enhances expressiveness and supports architecture reasoning in terms comparable to regular language [5]. Using the module and rule types understood by HUSACCT, we want to express architectural patterns.

HUSACCT supports the following types of software modules: subsystems, layers, components, interfaces and external systems. Subsystems are modules with clear responsibilities. A layer has the additional property of a hierarchical level, which enforces strict layering as HUSACCT automatically adds rules banning skip-calls and back-calls to the relevant levels. Components are modules whose contents are hidden behind and accessed through an interface module. Finally, external systems represent libraries, modules that are not actually part of the system under consideration [5] [6].

The rule types can be placed in one of two categories: property rule types and relation rule types [5]. The former consists of conventions such as naming and inheritance. Only the façade convention is used in this paper. This convention states that interface Module A always had to act as the interface for component Module B.

The second category, relation rule types, consists of a further subdivision into: "Is not allowed to use" and "Is allowed to use" rules, which are themselves two basic rule types that can be used to express rules like "Module A is not allowed to use Module B". Two rule types exist within the first subdivision (Back-call bans and Skip-call bans), which exist specifically for layer-type modules. Within the second subdivision, there are three: "Is only allowed to use", "Is the only module allowed to use" and "Must use". That makes a total of seven relation rule types, each tying two modules together and putting limitations on the dependencies allowed in the modular architecture. These rules are listed in Table I.

III. ARCHITECTURAL PATTERN DEFINITION

Architectural patterns, such as the N-Layered pattern (equivalent terms include Layered Architectures and N-Layers pattern), are not as precisely defined as the Gang-of-Four design patterns [8]. One can see them as two extremes on the same spectrum. The main characteristic of design patterns is that they are designed for solving recurrent problems on the level of the detailed design, e.g. source code, whereas architectural patterns exist on a system level. Consequently, design patterns appear more frequently within the same system, deal with far more specific concepts and are meant not to have any rule violations at all. Architectural patterns tend to focus more on which dependencies are *not* allowed. Design patterns specify the dependencies that *should* be implemented. As such, design patterns are more fit to be used as detailed design techniques similar to the tactics described by Bass et al. [1].

In this paper, we explore the possibility to express architectural patterns in terms of allowed and disallowed dependencies. Although at first sight this seems an easy exercise, in reality defining patterns this way leaves room for interpretation. Consider the 3-Layered pattern. This pattern is commonly used as a primary separation of concerns within a software system, e.g. user interface, business logic and data access layers. Each layer is only supposed to make use of its own internal modules or those of the layer directly below.

There are three layers and the following rules are how one would ordinarily express the pattern in terms of skip- and back-calls:

Rule set 1, Skip-calls and back-calls:

1) *Layer i is not allowed to skip-call to Layer $j > i + 1$.*
2) *Layer i is not allowed to back-call to Layer $j < i$.*

The fact that HUSACCT's layer-type modules possess hierarchical levels that constrain dependencies already suggests one way of defining the N-Layered pattern. However, we want to be able to use other module types as well, e.g. in order to use component-type modules as layers, which means we have to translate this set to one consisting of rules that do not rely on hierarchical levels.

A translation to "Is not allowed to use" rules would result in the following set of architectural rules for this pattern:

Rule set 2, "Is not allowed to use":

1) *Layer 2 is not allowed to use Layer 1.*
2) *Layer 3 is not allowed to use Layer 2.*
3) *Layer 3 is not allowed to use Layer 1.*
4) *Layer 1 is not allowed to use Layer 3.*

The same can be achieved by replacing rules 1 and 4 with "Is the only module allowed to use" rules from one layer to the next, or by two "Is only allowed to use" rules whilst removing rule 4. Both would appear to signify the exact same pattern. One might even prefer a hybrid form, such as the one displayed in rule set 3:

Rule set 3, Hybrid example:

1) *Layer 1 is the only module allowed to use Layer 2.*
2) *Layer 2 is only allowed to use Layer 3.*
3) *Layer 3 is not allowed to use Layer 1.*
4) *Layer 1 is not allowed to use Layer 3.*

Let us compare the different rule sets to see what happens when such a set is used as a pattern definition within a system's

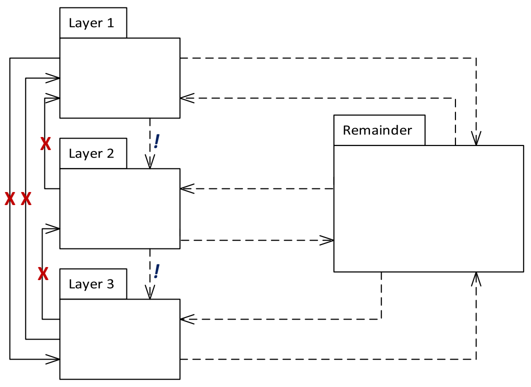

Fig. 1. The 3-Layered pattern and the Remainder, defined with “Is not allowed to use” rules (rule set 2) and showing allowed dependencies as dashed lines.

architecture. Imagine the rest of the architectural elements of this system as a single module, the *Remainder*, all architectural elements that are not part of the architectural pattern under consideration and not sub-modules of any of the pattern’s modules. As such, the Remainder is always with respect to some pattern. There may not be a Remainder if layering is the overall style of the system, but there has to be if it is a pattern used *within* system modules. More complex architectures, such as when there is a freely accessible additional module, may be found in a pattern discovery approach that takes a Remainder into account.

According to rule set 2, nothings prevents the Remainder from calling on any of the layers within the 3-Layered pattern and any of these layers can call upon the Remainder. This is depicted in Figure 1. Rule set 3 forbids any dependencies between the Remainder and Layer 2, as shown in Figure 2. This creates a scenario where the Remainder is free to call upon Layers 1 and 3, but not 2, while Layer 2 is the only layer that cannot call upon the Remainder. This version essentially isolates Layer 2 form the Remainder, while leaving a lot of freedom to the other two layers. In both cases, we have added “Must use” rules between the layers in order to enforce the correct usage of the pattern.

To graphically depict the patterns, we adopt a UML-like syntax. Expanded packages represent architectural elements, i.e. pattern modules and the Remainder. Collapsed packages indicate sub-modules. Legal dependencies are represented by dashed lines and directed associations figure as the architectural rules. Additional symbols are used to indicate the rule type being employed, as depicted in Table II. Conflicting rules result in exceptions. HUSACCT allows for rule exceptions by specifying the module that is exempt from the given rule [6].

At first sight, these rule sets seem to be equivalent. However, as an architectural pattern is an incomplete architecture, subtle differences emerge. Consider again the 3-Layered pattern.

More interpretations can be formulated using additional rule sets, combining the various rule types in different ways. We can thus identify the following interpretations of the N-Layered pattern:

TABLE II
THIS LEGEND PRESENTS THE DIFFERENT RULE TYPES FOR THE PROVISIONAL DIAGRAM NOTATION USED HERE.

“Is not allowed to use”	——X——>
“Is only allowed to use”	——O——>
“Is the only module allowed to use”	——|——>
“Is allowed to use”	- - - - - - - ->
“Must use”	- - - - ! - - - ->

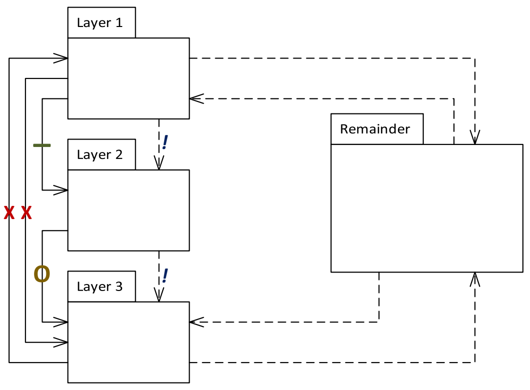

Fig. 2. The 3-Layered pattern and the Remainder, defined with three different rule types (rule set 3).

- ***N-Layered pattern (Complete freedom):*** *there are no restrictions with regard to the Remainder, hence the name. This corresponds to rule sets 1 and 2.*
- ***N-Layered pattern (Free Remainder):*** *the Remainder cannot be called by any layer, but can itself depend on any of them. The rule set requires two “Is only allowed to use” rules from each layer to the next and two “Is not allowed to use” rules for the back-call bans.*
- ***N-Layered pattern (Restricted Remainder):*** *the Remainder cannot call on any layer. This is with “Is the only module allowed to use” rules from each layer to the next and an “Is not allowed to use” rule from Layer 3 to 1.*
- ***N-Layered pattern (Isolated internal layers):*** *intermediary layers are never called by and can themselves not call on the Remainder. This corresponds to rule set 3.*

More varieties are conceivable, especially when different module-types are taken into account. This goes to show that even a pattern as simple as the N-Layered pattern allows for various different versions when the Remainder is taken into account. It shows that for pattern recognition, we need explicit pattern definitions. These same issues come up when one considers other architectural patterns and, for the sake of illustration, we will briefly go into one of them: the Model-View-Controller pattern.

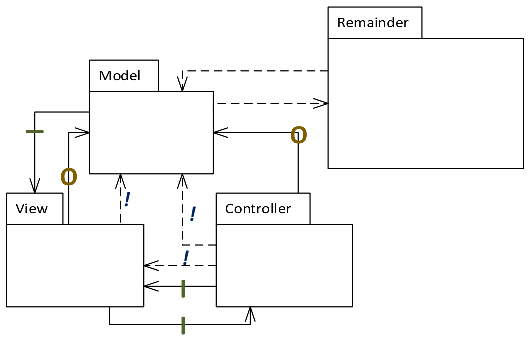

Fig. 3. The Model Interface interpretation, without demanding that Model talk to View.

Model-View-Controller

The Model-View-Controller (hereafter MVC) pattern is a term that applies to several related design patterns, but MVC also occurs on a system (architectural) level. It can be used to separate presentation logic from the underlying model logic by restricting communication between the two. A Controller acts as a mediator with the essential restriction that Model cannot use Controller [1]. The MVC pattern is often confused with the MVP (Model-View-Presenter) pattern [9] [10] and several others, but we are only interested in classic MVC for the sake of illustration. A common interpretation of the pattern states that View can perform a state query on Model and that Model can update View (e.g. Bass et al. [1]). We define the classic MVC pattern as follows:

Rule set 4, interpretation of classic MVC:

1) *Controller must use Model.*
2) *Controller must use View.*
3) *View must use Model.*
4) *Model is not allowed to use Controller.*
5) *If Model is not allowed to use View, violations should only be change updates.*
6) *If View is not allowed to use Controller, violations should all be triggered by user actions.*

It is clear that different interpretations of MVC would lead to several SRMA pattern definitions for this pattern as well. In fact, such ambiguity should be expected for all architectural patterns and this is precisely what we have to address in our research. Figure 3 graphically depicts one possible translation into HUSACCT's rules, namely the interpretation where Model forms the interface with the Remainder.

With these examples, we hope to illustrate the type of questions that are raised when defining architectural patterns in HUSACCT's SRMA terms. Not only are there various interpretations of patterns such as MVC, implementations of these patterns have consequences for the allowed, required and forbidden dependencies within a modular architecture. The nuances in interpretations can mostly be traced back to the dependencies with the Remainder. Patterns have an open world, which needs to be *made* explicit; whereas a complete architecture is closed and thus all rules *are* explicit. Assumptions and consequences will thereby become apparent.

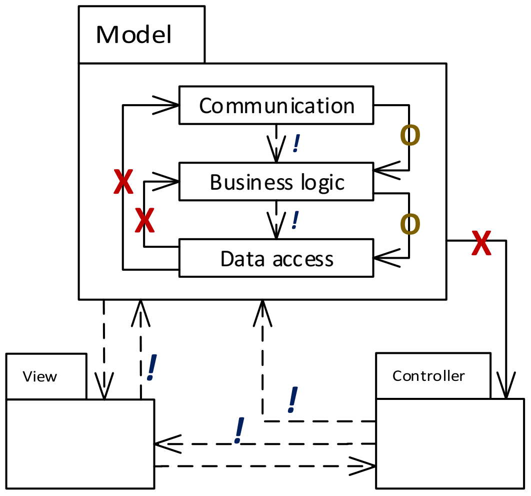

Fig. 4. A combination of Complete Freedom MVC (style) and Free Remainder 3-Layered (pattern), which fails to isolate the lower Model layers.

In the next section, we will discuss the effects of different pattern definitions on combinations of patterns.

IV. Pattern-based architectures

"Experienced architects typically think of creating an architecture as a process of selecting, tailoring, and combining patterns" [1]. An open-source example of this is SweetHome3D, where an MVC pattern is distributed over several layers [11]. We thus want to be able to recognise multiple patterns in the same architecture.

Although one would not expect to see a number of architectural patterns in the same order of magnitude as the number of design patterns within a given architecture, systems may include more than one architectural pattern. The exact interpretations of these patterns may result in subtle constraints or even contradictions. This makes explicit definition of patterns even more important, especially during a process like ACC or pattern recognition.

To illustrate this point, let us take the example of an architecture with an MVC pattern. Figure 4 shows a different interpretation of MVC from Figure 3, namely the "Complete Freedom" variety. This interpretation uses only the "Is not allowed to use" rule between Model and Controller besides its "Must use" rules. While MVC is the overall style, there is a 3-Layered pattern within the Model module. This is quite reasonable and plausible, as Model might contain business logic and data access functionality separated from the rest (let us call Layer 1 "communication") through layering. This is also part of the reason why we chose to discuss the MVC pattern earlier. It begs the question, however, of who can access whom in this case.

Based on the architectural rules of these two patterns alone, are the lower layers of Model allowed to have dependencies to and from View and Controller? And, if there happens to be any, what about the Remainder outside the MVC pattern? From the perspective of this 3-Layered pattern, View and Controller are part of the Remainder and therefore the particular pattern definition has a subtle effect. If we were to apply the “Isolated Internal Layers” version of the N-Layered pattern (Figure 2), the data access layer would be allowed to have dependencies to and from View and Module. The “Free Remainder” interpretation used in Figure 4 would allow dependencies of these outside modules with the data access (in both directions) and business logic (one way) layers, which may also not be desirable.

Probably, the architect wanted to isolate the lower layers from the other MVC modules. Thus, the “Restricted Remainder” interpretation (two “Is the only module allowed to use” rules and one “Is not allowed to use”) ought to be combined with rules that prohibit the data access layer from accessing anything but those modules that it should require. This may be solved with some of the relational rules, or by turning Model into a component type with either the communication layer or an additional element within Model functioning as the interface for this component, thereby calling upon the façade convention that was briefly mentioned in Section 2.

If these combinations raise such questions, then pattern recognition within pattern-based architectures needs to take into account that identifying overall style first could result in a different final conclusion from an approach to pattern recognition starting with the smaller pattern. We are currently investigating this problem with our pattern discovery approach.

V. Related Literature

Several languages have been used to define architectural patterns for various purposes by other researchers. Abowd, Allen and Garlan rely on the Z language, a mathematical notation based on predicate logic, to introduce styles with precise syntactic and semantic descriptions of both static and dynamic characteristics [12]. This is more formal than what we have needed thus far.

Another common language is the Acme Architecture Description Language (ADL). It is used in AcmeStudio, for example, during the design phase of systems and allows for new styles to be defined by the user [13]. Although we too are interested in user-defined patterns, we hope to avoid the need for predicate logic on the part of the user. Research involving Acme has produced several publications, such as one on the translation of Acme into Alloy and subsequent property analysis in the Alloy Analyzer [14], as well as an ontology of styles [15].

Not all researchers base themselves on predicate logic. Sartipi presents an extensive approach to SAR based on patterns and data mining that makes use of both clustering and editing-cost-based graph matching techniques [16]. Although this has a strong mathematical background and promising results, it requires substantial user input, but this may well be unavoidable for SAR. Runtime analysis is also used to find patterns, such as by the DiscoTect system, which employs specifically tailored state machines [17].

For a systematic literature review on architecture reconstruction, we refer the reader to the 2009 publication by Ducasse and Pollet [18], in which they mention architectural patterns and styles several times in their attempts to build an ontology of various methods.

There exist several pattern discovery approaches for the purposes of finding design patterns. These do not necessarily apply to patterns at the system level, but their ideas are useful to us nonetheless. Techniques used here rely, for example, on class metrics [19] and heuristics such as inheritance relationships [20]. A systematic literature review on design pattern mining was published in the same year as the aforementioned SLR [21].

VI. Industrial Impact

Our research is still in an early stage, but we believe that it is promising. Obviously, stored pattern definitions within an ACC tool make it easier for users to specify the intended architecture through its (graphical) interface. Similarly, a set of predefined patterns aids the design process. Not only in terms of convenience, but also to aid the discussion about pattern interpretation that is commonly required during architecture design.

More interestingly, the actual reason we are concerned with this topic is that both ACC and SAR would benefit from a pattern discovery process using ACC tools. Some 80% of all costs in software development are related to maintenance activities [1] and architecture documentation can improve maintainability. ACC and SAR can be seen as two sides of the same coin. In both cases, hypothesised portions of the architecture (patterns or otherwise) have to be identified. If this hypothesis is derived from documentation, one is talking about ACC. Part of our research is focussed on using code and dependencies to derive the module view, albeit but a tool in a larger tool box. Dynamic analysis as well as additional views should all come into play in a complete reconstruction method. Considerations of specifying patterns in tool-specific, preferably semantically rich, terms, are necessary. The ability to find a hypothesised piece of architecture would greatly speed up the reconstruction process. Studying dependencies that break pattern rules is one thing, but the choice which pattern instances to look for requires algorithm design. This is the focus of our ongoing research.

We envision future software tools based on ACC functionality that would allow users to request a search for any partial architecture, whether it be a common pattern or a user-defined piece of architecture. Using dependencies, but perhaps also semantic constraints and clustering techniques, such an approach can help anyone trying to piece together an existing architecture, which is a difficult task. Improvement in the speed at which this can be performed makes it much easier to model existing systems or support the development process, although checking dependencies becomes itself a slow process with large systems. HUSACCT is not a widely adopted tool and is limited to Java and C# code, but it is useful for academic purposes. In the future, we want to be able to incorporate other languages and combinations within the same system. We are currently working on pattern discovery algorithms and their evaluation by means of case studies.

During these case studies, we also intend to look into adoption issues. ACC/SAR tools are always constrained in terms of views and artefacts, so any approach will have its limitations. Integration with other techniques and tools is therefore essential if one wants to provide organisations with an attractive solution. Accordingly, we believe even a mature version of our approach should not stand on its own, but should come with a range of complementary approaches.

VII. Conclusions

With this paper, we show how HUSACCT's architectural elements allow one to think about architectural pattern definitions in some precision. This raises questions such as what we mean by a particular pattern or what layering implies with regard to elements that are not part of any layer. These questions and the choices of definitions that follow from them may be useful for architects to describe exactly what they mean by their applied patterns, but it can also be fruitful for purposes other than software design. Our own current and future research is focussed on the usage of these SRMA pattern definitions in ACC and SAR, where we want to detect instances of architectural patterns within existing architectures. The pattern definitions we choose and the order in which we detect such patterns can have profound effects on the allowed dependencies. It is therefore imperative that we make all rules, assumptions and consequences surrounding architectural patterns completely explicit.

Acknowledgement

We thank our colleagues at Utrecht University, with special thanks to Leo Pruijt of the HU University of Applied Sciences for his many contributions and insights during our research. We also thank the reviewers for their constructive criticism.

References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., P. Gordon, Ed. Boston, Massachusetts, USA: Addison-Wesley Professional, 2012.

[2] N. Rozanski and E. Woods, *Software Systems Architecture: Working with stakeholders using viewpoints and perspectives.*, 2nd ed. Addison-Wesley, 2011. [Online]. Available: http://www.viewpoints-and-perspectives.info/

[3] R. N. Taylor, N. Medvidović, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-EHEP000180.html

[4] J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," in *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*. IEEE, jan 2007, pp. 12–12. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4077029

[5] L. Pruijt and S. Brinkkemper, "A metamodel for the support of semantically rich modular architectures in the context of static architecture compliance checking," in *Proceedings of the First International Conference on Dependable and Secure Cloud Computing Architecture - DASCCA '14*. New York, New York, USA: ACM Press, 2014, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2578128.2578233

[6] L. Pruijt, C. Köppe, J. M. E. Van der Werf, and S. Brinkkemper, "HUSACCT: Architecture Compliance Checking with Rich Sets of Module and Rule Types," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*. Vasteras, Sweden: ACM, 2014, pp. 851–854. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2642937.2648624

[7] L. Pruijt, C. Köppe, and S. Brinkkemper, "On the accuracy of Architecture Compliance Checking Support," in *IEEE International Conference on Program Comprehension*, San Francisco, CA, USA, 2013, pp. 172–181. [Online]. Available: http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=6613845&abstractAccess=no&userType=inst

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=186897

[9] M. R. J. Qureshi and F. Sabir, "A Comparison of Model View Controller and Model View Presenter," *CoRR*, vol. abs/1408.5, 2014. [Online]. Available: http://arxiv.org/abs/1408.5786

[10] M. Potel, "MVP : Model-View-Presenter The Taligent Programming Model for C++ and Java," Taligent, Inc., Tech. Rep., 1996. [Online]. Available: metrology.googlecode.com/svn-history/r350/trunk/doc/ebooks/mvp.pdf

[11] E. Puybaret, "SweetHome3D," 2015. [Online]. Available: http://www.sweethome3d.com/

[12] G. D. Abowd, R. Allen, and D. Garlan, "Using Style to Understand Descriptions of Software Architecture," in *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*. Los Angeles, CA, USA: ACM, 1993, pp. 9–20. [Online]. Available: http://dl.acm.org.proxy.library.uu.nl/citation.cfm?id=167055

[13] B. Schmerl and D. Garlan, "AcmeStudio: supporting style-centered architecture development," in *26th International Conference on Software Engineering*. IEEE, 2004, pp. 704–705. [Online]. Available: http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=1317497&abstractAccess=no&userType=inst

[14] J. S. Kim and D. Garlan, "Analyzing architectural styles," *Journal of Systems and Software*, vol. 83, no. 7, pp. 1216–1235, jul 2010. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0164121210000336

[15] C. Pahl, S. Giesecke, and W. Hasselbring, "Ontology-based modelling of architectural styles," *Information and Software Technology*, vol. 51, no. 12, pp. 1739–1749, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2009.06.001

[16] K. Sartipi, "Software architecture recovery based on pattern matching," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE, 2003, pp. 293–296. [Online]. Available: http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=1235434&abstractAccess=no&userType=inst

[17] H. Y. H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, "DiscoTect: a system for discovering architectures from running systems," in *Proceedings. 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 470–479. [Online]. Available: http://dl.acm.org.proxy.library.uu.nl/citation.cfm?id=999450

[18] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, 2009. [Online]. Available: http://www.computer.org/csdl/trans/ts/2009/04/tts2009040573-abs.html

[19] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Design pattern recovery in object-oriented software," in *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*. Ischia, Italy: IEEE, 1998, pp. 153–160. [Online]. Available: http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=693342&abstractAccess=no&userType=inst

[20] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 896–909, 2006. [Online]. Available: http://ieeexplore.ieee.org.proxy.library.uu.nl/xpl/freeabs_all.jsp?arnumber=4015512&abstractAccess=no&userType=inst

[21] J. Dong, Y. Zhao, and T. Peng, "a Review of Design Pattern Mining Techniques," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 06, pp. 823–855, 2009. [Online]. Available: http://www.worldscientific.com.proxy.library.uu.nl/doi/abs/10.1142/S021819400900443X