A Genetic Approach to Architectural Pattern Discovery

Joeri Peters, Jan Martijn E.M. van der Werf
Department of Information and Computing Sciences
Utrecht University
Princetonplein 5, 3584 CC Utrecht, The Netherlands
{j.g.t.peters, j.m.e.m.vanderwerf}@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 realised artefacts adhere to the architecture. Typically, these techniques require a complete architecture as input.

In this paper, we present a genetic approach to express and discover architectural patterns based on the allowed and disallowed dependencies between the pattern elements. Through static ACC, we validate the genuineness of the found instances. Initial validation shows the potential of the approach.

CCS Concepts

•**Software and its engineering** → **Software reverse engineering;** •**Computing methodologies** → *Genetic algorithms;*

1. INTRODUCTION

Technical debt has many different causes, one of them being outdated architectural documentation [2]. Due to various reasons, realised software units (SUs) drift apart from the intended architecture, thus creating architectural erosion [13]. For many Software Producing organisations (SPOs) [3] this is a clear risk, as they need to update and improve their software products constantly to stay in business. However, due to many external constraints, such as time-to-market, proper architectural documentation comes at the bottom of the list. Consequently, many of these organisations have outdated documentation [12]. Thus the risk of technical debt is high for such organisations [25].

Architecture Compliance Checking (ACC) has the potential to support organisations in maintaining up-to-date documentation by determining whether the realised SUs adhere to the documented architecture [10], often referred to as the intended architecture. Current state-of-the-art ACC techniques base themselves on a given mapping between SUs and (static) architectural elements (AEs), and construct a dependency graph from the SUs to check whether there are no violating dependencies [6,19]. Many of the software products delivered by SPOs can be categorised as Very Large Software Systems (VLSSs) [27], for which this approach has several important drawbacks that prevent these techniques from being used in practice [19], including:

1. Insufficient architectural documentation;
2. The mapping between SUs and AEs is infeasible; and
3. The dependency graphs become too large to analyse.

Instead of creating a complete architectural definition, developers adopt a certain architectural style [24], defining allowed and disallowed dependencies. Hence, mapping all SUs to AEs is not possible since (a) the appropriate AEs are not defined, and (b) as VLSSs contain many millions of lines of code, so the number of elements to map becomes too large. In case no appropriate architectural documentation exists, Software Architecture Reconstruction (SAR) can be used to derive documentation [4] [11]. Many SAR techniques base themselves on the dependency graph, which again becomes infeasible for very large software products. Another issue with SAR is to determine the appropriate level of abstraction to be of value for VLSSs.

The contribution of this paper is twofold. First, we propose a definition of architectural patterns in terms of allowed and disallowed dependencies (Section 3). This gives the architect the opportunity to specify the architectural solution and constraints on a high level, without having to specify all details of the design. Second, we present a genetic algorithm that searches for genuine instances of these architectural patterns among the realised software units (Section 4). A genuine instance of a pattern is defined as a mapping between SUs and AEs such that the number of violations against the pattern is minimal. We show the applicability of the approach in Section 5. Section 6 concludes the paper.

2. COMPLIANCE & RECONSTRUCTION

Software architecture entails the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both [2]. One can distinguish three types of structure within software architecture (static, dynamic and allocation) [2], or look at a categorisation based on views and viewpoints, such as functional, development, and concurrency [21]. This paper is re-

ECSAW '16, November 28-December 02, 2016, Copenhagen, Denmark

ISBN 978-1-4503-4781-5/16/11…$15.00

DOI: http://dx.doi.org/10.1145/2993412.3003393

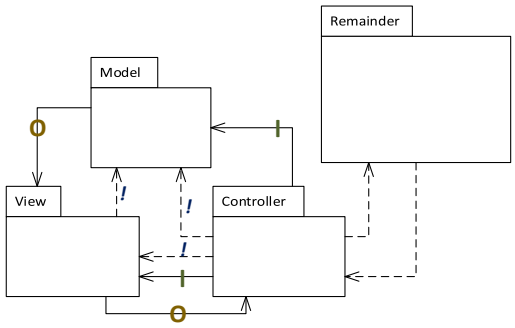


Figure 1: The MVC pattern where interaction with the Remainder is only via the Controller.

stricted 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).

Architecture Compliance measures to which degree the realised software units conform to the intended architecture [10]. ACC intends to bridge the gap between the high-level abstraction of the software architecture and the realised software. Static ACC focuses on the static aspects of the architecture.

Most static ACC approaches build upon dependency analysis: "the process of determining a program's dependencies" [17]. ACC tools like HUSACCT [20] check whether the realised dependencies do not violate the dependencies defined by the architectural documentation.

ACC can only be applied if some architectural documentation exists. In case no such documentation exists, Software Architecture Reconstruction (SAR), tries to re-engineer the architecture of an already realised system. It is "an interpretive, interactive, and iterative process involving many activities; it is not automatic" [2]. In other words, SAR focuses on understanding the software architecture of a software system from the realised software units, which is rather difficult to do even if the system is relatively small.

For a systematic literature review on architecture reconstruction, we refer the reader to [6].

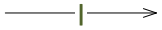
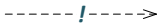
In essence, SAR and ACC are two sides of the same coin: in ACC, the existing architectural documentation can be seen as the hypothesis to validate on the implemented software units, whereas in SAR a hypothesis is generated from the software units and then validated. This observation is key to the approach we present in this paper.

3. ARCHITECTURAL PATTERNS

In larger software systems, architects typically adopt a certain architectural style [24], such as a layered design, a Model-View-Controller or a Service-Oriented Architecture. Such styles guide the architects in the system design. Choosing a style means that the design applies some architectural pattern. Such 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 [2].

Architectural patterns are not as precisely defined as the Gang-of-Four design patterns [7]. The main characteristic of design patterns is that they are designed for solving re-

Table 1: This legend presents the different rule types for the provisional diagram notation used here.

	"Is allowed to use"
	"Is not allowed to use"
	"Is only allowed to use"
	*"Is the only module allowed to use"*
	*"Must use"*

current 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. [2].

Inspired by the definition of Semantically Rich Modular Architectures [18], we define a dependency-based architectural pattern language. It consists of architectural elements, the special element called the *Remainder*, and 5 dependency rule types, shown in Table 1. Additionally, rules can have exceptions. The *Remainder* is an element used to define how the pattern interacts with architectural elements not touched by the pattern. It is a module outside of the pattern and thus represents the rest of the architecture.

As an example, consider the MVC pattern depicted in Figure 1. Not that, in this depiction, conflicting rules imply exceptions to those rules, which allows us to use these specific rule types. It can be specified with three architectural elements: the Model, the View and the Controller. To specify that communication with other architectural elements can only be performed via the Controller, we add two rules that specify that the Remainder can be used in the Controller, and that the Remainder can use the Controller. This results in the following set of rules:

1. *Controller must use Model.*
2. *Controller must use View.*
3. *View must use Model.*
4. *Model is not allowed to use Controller.*
5. *Model is not allowed to use View.*
6. *View is not allowed to use Controller*

Rules 5 and 6 often have exceptions for change updates and user actions, respectively, in practice. The above set

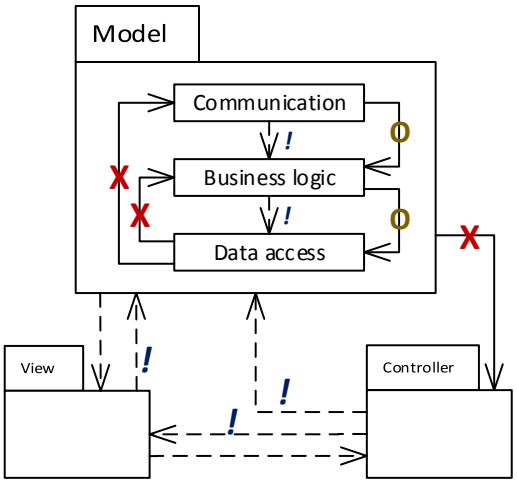


Figure 2: A combination of the MVC pattern and a 3-Layered pattern, which fails to isolate the lower Model layers.

implies that all communication with the Remainder happens via the Controller.

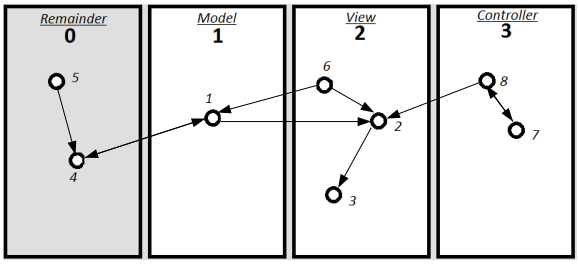
Making the interaction with Remainder explicit allows us to combine patterns, and to reason about their interactions. As an example, Figure 2 shows an architecture containing two patterns: the main pattern is the MVC pattern, where the Model uses a 3-Layered pattern. Notice that, as the 3-Layered Pattern uses the "is only allowed to use" rules, the Remainder, i.e. the View and Controller elements, are allowed to use elements within each of the 3 layers. For a more elaborate discussion on the pattern language, we refer the reader to [14] [15].

4. GENETIC DISCOVERY OF PATTERNS

The advantage of defining a partial architecture in terms of allowed and disallowed dependencies, is that we can use existing ACC tools to validate whether the realised software units adhere to the patterns. Consequently, if we provide a mapping of software units to the architectural elements of the pattern, the ACC tools provide information about how many and which dependencies are violated, and how many are adhered to. This gives us valuable insights in whether the mapping adheres to the pattern.

In this way, ACC can assist us in finding a best mapping between software units and architectural elements: by testing each possible mapping, we obtain a best match. For instance, software dependencies from a UI layer to a business logic layer comply perfectly with the traditional 3-layered architecture; but the reverse, dependencies from the middle to the upper layer, is not allowed in the case of strict layering.

Based on this observation, we propose a genetic approach to discover genuine instances of an architectural pattern. Based on how well the mapping fits, we can define mutations of and cross-overs between different mappings to guide the search towards a best candidate mapping. We base ourselves on the open-source tool HUSACCT [20]. Its ACC functionality allows us to judge whether software dependencies derived from source code are legal according to a specific pattern. The output of HUSACCT can then be used to define the fitness of the mapping compared to the architectural pattern.



Gene (software unit)	1	2	3	4	5	6	7	8
Allele (pattern module)	1	2	2	0	0	2	3	3

Figure 3: The mapping of a pattern candidate

4.1 Fitness

In order to compare several mappings with each other, it is necessary that there be a fitness score. Such a score can express the "goodness" of a particular mapping of software units (groupings of source code) to pattern modules. To calculate this score, a fitness function has to be defined as part of our discovery algorithm. To our knowledge, there is no single function that would be ideal for this purpose and we have considered several formulae for this purpose. One function might tend towards one type of solution, such as large patterns with few dependencies, another function might have a bias towards another, like small patterns with low coupling.

We selected a fitness function inspired by the F-measure (the harmonic mean of precision and recall [23]), as we also have two measures to optimise. Our variables, the values of which are determined by HUSACCT, are as follows: $D \in \mathbb{N}$ is the number of dependencies between the pattern modules and with the Remainder, $M \in \mathbb{N}$ is the number of dependencies between pattern modules that are in line with that pattern's "Must use" rules (these are essential dependencies, since these are the ones that are explained by the existence of the pattern) and $V \epsilon \mathbb{N}$ is the number of violating dependencies. Violation of a "Must use" rule implies that $M = 0$ for that particular rule.

There are two ratios to be optimised. One is the number of dependencies explained by the pattern relative to the total number of dependencies of that candidate ($\frac{M}{D}$), which should be maximised. The other is the number of violations relative to the number of dependencies that are not explained by the pattern ($\frac{V}{D-M}$).

$$
\begin{aligned}
f(D,V,M,\beta) &= (1+\beta^2)\frac{(1-\frac{V}{D-M})\cdot\frac{M}{D}}{\beta^2\cdot(1-\frac{V}{D-M})+\frac{M}{D}} \\
f(D,V,M,1) &= 2\frac{(1-\frac{V}{D-M})\cdot\frac{M}{D}}{(1-\frac{V}{D-M})+\frac{M}{D}}
\end{aligned}
\tag{1}
$$

The variable $\beta \epsilon \mathbb{R}$ controls the relative importance of one ratio over the other by controlling the relative weight of one expression in the mean. To establish this fitness function with both ratios equally valuable, take $\beta = 1$. Since we have no reason to emphasise one measure over the other, this is the value we chose.

During the evaluation of this function, we apply a heuris-

tic that states that a violation of a "Must use" rule, i.e. when there are no dependencies between pattern modules that ought to be connected, the fitness is automatically minimised. Architectural patterns should explain certain dependencies, so their absence is a dead give-away that a particular candidate is unlikely to be a genuine instance of that pattern.

4.2 Brute Force Search Space

Given a mapping between software units and architectural elements, we can use the output of HUSACCT to calculate a score using the aforementioned fitness function. What remains is an algorithm for the discovery process. As a baseline approach, we take an exhaustive search of all possible mappings of software units to pattern modules, the brute force approach.

The number of pattern candidates, i.e. the different mappings, grows rapidly with the number of software units n and the number of architectural elements k. Exactly how rapid depends of two factors: Aggregation and Remainder.

Aggregation allows multiple software units to be mapped to the same architectural element within the pattern. For example, if software units are merely Java code classes, it is more than reasonable to assume that multiple software units ought to be mapped to a particular architectural element of a pattern, such as a layer. If the software units are grouped in e.g. packages or namespaces, these can be used as software units, instead of the individual classes. This limits the search space drastically.

As explained in the previous section, the Remainder is used to represent those software units that are not mapped to any specific architectural element within the pattern. It is useful to allow for such a classification, particularly if the architectural pattern is not intended to be the completely encompassing structure of the system under analysis [14]. Especially for larger systems in which a pattern is applied on a subset of the software units, it is beneficial to allow for a Remainder. Unfortunately, having a Remainder has the drawback that the algorithm allows for mappings in which some of the software units are not mapped. The exact definition of architectural patterns using the tool-specific language can run into ambiguities due to this Remainder. For a more elaborate discussion on the function of the Remainder, see [14].

Given the values of n and k, the number of candidates equals in the non-aggregation case:

$$N(n,k) \quad = k! \cdot \binom{n}{k} \tag{2}$$

The existence of a Remainder is simply implied if $n > k$. Allowing for aggregation implies a faster growth of the number of pattern candidates to be evaluated in a brute force approach, namely according to:

$$N^a(n,k) \quad = k! \cdot \left\{ {n \atop k} \right\} = \sum_{j=0}^{k} (-1)^{k-j} j^n \tfrac{k!}{j!(k-j)!} \tag{3}$$

where $\left\{ {n \atop k} \right\}$ denotes the Stirling Number of the Second Kind, the number of ways n objects can be distributed over k unlabelled baskets.

Finally, allowing for a Remainder in addition to aggregation results in an even more explosive expression:

$$\begin{aligned} N^{a,r}(n,k) &= k! \cdot \left\{ {n \atop k} \right\} + (k+1)! \cdot \left\{ {n \atop k+1} \right\} \\ &= \sum_{j=0}^{k} \left((-1)^{k-j} \tfrac{j^n}{j!} \left(\tfrac{k!}{(k-j)!} - \tfrac{(k+1)!}{(k+1-j)!} \right) \right) + (k+1)^n \end{aligned} \tag{4}$$

Table 2: The number of pattern candidates for various values of n when $k = 3$, in all three cases.

n,k	N	N^a	$N^{a,r}$
3,3	6	6	6
4,3	24	36	60
5,3	60	150	390
6,3	120	540	2,100
7,3	210	1,806	10,206
8,3	336	5,796	46,620
9,3	504	18,150	204,630
10,3	720	55,980	874,500
15,3	990	14,250,606	1,030,793,406
20,3	1,320	3,483,638,676	1,089,054,420,300

In order to provide an illustration of how quickly these numbers grow to be unwieldy, consider Table 2. This table suggests that a brute force approach to our ACC-based pattern discovery method may be a reasonable solution only when dealing with relatively low numbers of n and $k = 3$. Since many architectural patterns consist of only a few modules, typically three (3-Layered, MVC, Broker pattern), it strongly depends on the number of software units whether this approach can be feasible in a reasonable amount of time. Regardless, though, such a process would still be faster than any human could do the same by hand, thereby already making it a useful solution despite its inefficiency.

4.3 Genetic Algorithm

After completing the brute force approach as a baseline algorithm, we proceeded to produce a more sophisticated approach. One would expect that several of the myriad of pattern candidates that are considered in a brute force search are actually hopelessly poor candidates. Evaluating the fitness score for such candidates is essentially wasteful, although this seems unknowable beforehand. However, as an attempt to circumvent this issue, we implemented a rudimentary genetic algorithm for this pattern discovery process. Genetic algorithms, a subset of evolutionary algorithms, allow for the optimisation of specific search problems by relying on heuristics aimed at mimicking natural selection, finding a useful solution to a problem by allowing some kind of genetic code to evolve [8]. This requires a population of individuals, programmed chromosomes that code for specific solutions to the problem at hand. Using our fitness function, we allow some of these chromosomes to reproduce and randomly mutate. It makes more sense to attempt pattern candidates that are similar to those with good fitness scores, since shuffling a few software units around might produce an even better solution. This can eventually lead to local optima and possibly the global optimum in a shorter timespan than an exhaustive brute force search would. Hill-climbing algorithms that do not take this genetic approach are generally more likely to get stuck in such a local optimum.

Our chromosomes are simple arrays of integer genes, the values (alleles) of which refer to pattern modules. The first integer refers to the mapping of the first software unit, the

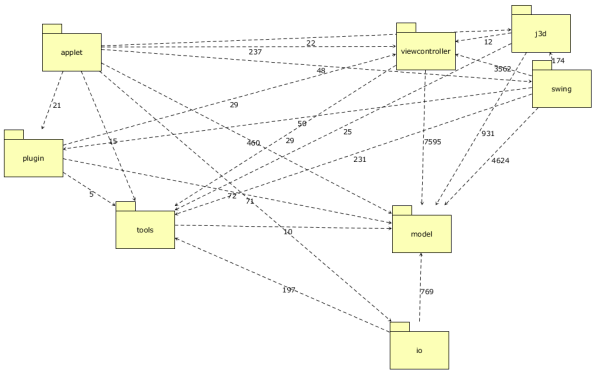


Figure 4: The Java packages in the root of SweetHome3D.

second integer to the mapping of the second software unit, and so on. An integer value of 0 means the software unit is assigned to the Remainder, a value of 1 means an assignment to pattern module 1, etc. The reverse, genes referring to pattern modules and their allele-values referring to software units, may seem more intuitive, but this would result in an unnecessary increase in the complexity of the genes' data type when these chromosomes are to allow for aggregation.

Figure 3 shows the relation between a chromosome and the pattern candidate for which it is a representation. The chromosome describes a mapping of several software units, with dependencies linking them to one another, to the modules of an MVC-pattern. Evidently, this is a case where both aggregation and a Remainder are allowed. The dependencies between the software units may or may not be in accordance with the specified rules of the MVC-pattern. Therefore, this chromosome has a particular fitness score. Changing on of the integer values in the chromosome changes the mapping of the software units, thereby creating a new pattern candidate.

An existing library for genetic algorithms, JGAP (Java Genetic Algorithms Package) was chosen, due to its modular nature and accessible documentation[1]. JGAP provides a basic genetic framework in which only the essentials have to be implemented: the fitness function and the specification of chromosomes. JGAP requires these two classes to be written, but allows for many more aspects of its configuration to be changed or expanded. The fitness function has been implemented using HUSACCT. The specification of chromosomes represents the mapping between software units and architectural elements. JGAP provides basic support for mutations and cross-overs in the form of random mutations and cross-overs within the population. More sophisticated mutations and cross-overs based on the output of HUSACCT are possible, but remain future work.

5. INITIAL EVALUATION

To evaluate our approach, we applied it to the open-source Java application SweetHome3D. Its package hierarchy, depicted in Figure 4, shows eight root packages. None of them have any sub-packages, although each of them has a pletora of individual classes. Based on the developer's forum we distill that its architecture is a combination of 3-Layered with MVC [2]. The overall style is a 3-Layered architecture in which the middle layer is called by both of the other layers, combined with a specialization of the MVC pattern: the Model-Viewcontroller pattern where the View and Controller are contracted, and Model does not have any outgoing dependencies) distributed over the business logic and presentation layer. The layered pattern is not difficult to find because of the relatively large numbers of dependencies along the lines of the "Must use" rules. A high number of "Must use" affirmations results in good fitness score, as it explains all these dependencies with the existence of a pattern.

Figure 5 shows the output of the genetic algorithm to find the centralised layering described above. It depicts a solution similar to the one we anticipated (except for the placement of Tools). The main difference is that Layer 1 is Layer 3 and vice versa. As Layer 1 and Layer 2 are independent, because of the way the pattern is defined, this does indeed seem just as reasonable. Without semantic constraints, this

```
End evolution iteration 29
Total evolution time: 424834 ms
Presenting the ordering of software units mapped in the
chromosomes:

1: com.eteks.sweethome3d.applet
2: com.eteks.sweethome3d.io
3: com.eteks.sweethome3d.j3d
4: com.eteks.sweethome3d.model
5: com.eteks.sweethome3d.plugin
6: com.eteks.sweethome3d.swing
7: com.eteks.sweethome3d.tools
8: com.eteks.sweethome3d.viewcontroller

The best (and unique) chromosomes are printed here. If their
number is particularly small, it is because the population
contained many duplicates.
This would mean that the algorithm has converged on a small
number of solutions. These should indicate at least local
optima, if not the global optimum.

Chromosome 1:
01320323
Fitness value: 0.971235879278368
Placing best candidate in defined architecture...
Best candidate was successfully mapped and validated.
Elapsed time: 424.991 seconds.
```

Figure 5: A genetic (with Remainder) run on SweetHome3D's packages for the Centralised Layering pattern.

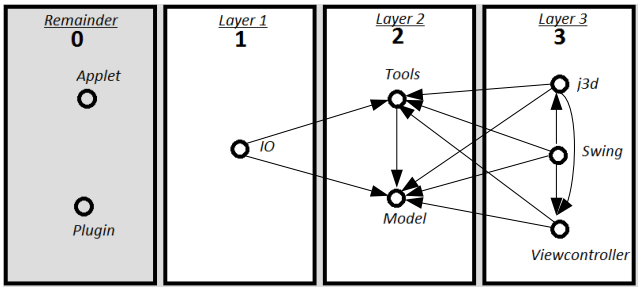


Figure 6: The best candidate discovered (GA) for the centralised layering architecture. Dependencies with the Remainder are excluded.

[1]The JGAP website: http://jgap.sourceforge.net/.

[2]http://www.sweethome3d.com/support/forum/viewthread_thread,3067

```
End evolution iteration 29
Total evolution time: 109273 ms
Presenting the ordering of software units mapped in the
chromosomes:

1: com.eteks.sweethome3d.applet
2: com.eteks.sweethome3d.io
3: com.eteks.sweethome3d.j3d
4: com.eteks.sweethome3d.model
5: com.eteks.sweethome3d.plugin
6: com.eteks.sweethome3d.swing
7: com.eteks.sweethome3d.tools
8: com.eteks.sweethome3d.viewcontroller

The best (and unique) chromosomes are printed here. If their
number is particularly small, it is because the population
contained many duplicates.
This would mean that the algorithm has converged on a small
number of solutions. These should indicate at least local optima,
if not the global optimum.

Chromosome 1:
21212210
Fitness value: 1.0
Placing best candidate in defined architecture...
Best candidate was successfully mapped and validated.
Chromosome 2:
20010201
Fitness value: 1.0
Chromosome 3:
20001010
Fitness value: 1.0
Chromosome 4:
20110010
Fitness value: 1.0
Chromosome 5:
21200010
Fitness value: 1.0
Chromosome 6:
21211000
Fitness value: 1.0
. . .
Chromosome 69:
21000200
Fitness value: 1.0
Elapsed time: 109.399 seconds.
```

Figure 7: Partial output of a genetic run on SweetHome3D's packages for the Model-View-Controller pattern.

sort of confusion is to be expected. The mapping is illustrated in Figure 6.

The MVC pattern is more difficult to find, due to the large number of equally good candidates. Without semantic input, it is very difficult to discover the pattern. As the output, of which an excerpt is shown in Figure 7, shows, there are many candidates that satisfy the criteria of the specialised MVC pattern. To select the appropriate candidates, the semantic or nomenclature of the software units need to be considered as well.

Defining novel patterns is essential to our approach, because any partial architecture can be thought of as a "pattern" and subsequently used in the same way. When we combine the two found architectural patterns and search for the result, we obtain the architecture as depicted in Figure 8.

6. CONCLUSIONS AND FUTURE WORK

Architectural patterns and styles allow the architect to partially describe an architecture. In this paper, we presented an approach to define architectural patterns in terms of allowed and disallowed dependencies. This allows us to use ACC tools to discover whether a set of realised software units adhere to some given architectural pattern. We define a fitness function that calculates the score of a mapping between software units and the architectural elements within a pattern. Using a genetic algorithm, we explore the search space of all possible mappings to identify genuine instances of the pattern. The approach is implemented using JGAP and HUSACCT. Initial validation on the open-source Java application SweetHome3D shows the potential of the approach.

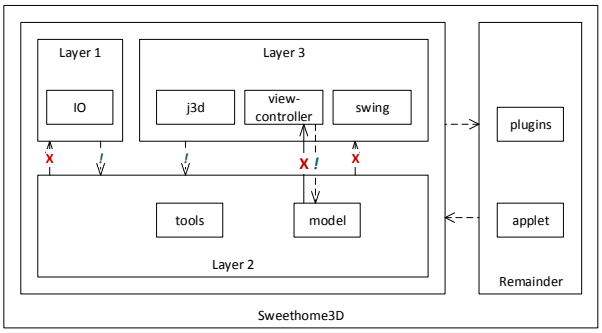


Figure 8: Sweethome3D's architecture

SAR based on dependency analysis is not new. Examples include a pattern-based clustering and graph matching technique [22], as well the state machine-based DiscoTect system [28]. Architectural style representations have been used in reverse engineering using style and query libraries [9]. An framework has been extended by including patterns (architectural, design and code patterns). This relies on "hotspots", heuristic indicators of a pattern [16].

Methods that are primarily aimed at design patterns must not be neglected, even if problems are to be expected on a system level. Methods exist based on class metrics [1] and heuristics like inheritance [26]. We refer the reader to two SLRs: [5] and [6].

The genetic approach chosen in this paper has several drawbacks. For example, parametric architectural patterns, such as the N-layered pattern, or the Peer-to-Peer pattern cannot be expressed using the proposed dependency-based architectural pattern language. Due to the unknown number of architectural elements, the metaphor of genetic algorithms cannot be applied.

By defining specialised mutations and cross-overs, the population generation can be guided. For example, based on the ACC outcomes, not only the complete mapping can be evaluated, also parts of the mapping can be compared. A simple crossover would take from the two mappings the best elements, and create a new mapping from those. Similarly, mutations can be defined using the outcomes of the ACC tool. Defining more advanced mutations and crossovers remain future work for now.

Another element for future work is the fitness function. As it only considers dependencies, genuine instances can be found that from a semantic point of view make no sense. Hence, the fitness function should take semantics and nomenclature of the software units into account to obtain more useful results.

Initial evaluation shows the potential of the approach. However, as the proof of the pudding is in the eating, we require tool support for the architects to define their own architectural patterns to be able to evaluate the approach in larger case studies.

Acknowledgements

The authors would like to thank Leo Pruijt and Jurriaan Hage for the fruitful discussions, and the anonymous reviewers for their valuable feedback and suggestions.

7. REFERENCES

[1] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *, 6th International Workshop on Program Comprehension, 1998. IWPC '98. Proceedings*, pages 153–160, June 1998.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Series in Software Engineering. Addison Wesley, Reading, MA, USA, 2012.

[3] S. Brinkkemper and X. Lai. Concepts of product software. *European Journal of Information Systems*, 16(5):531–541, 2007.

[4] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[5] J. Dong, Y. Zhao, and T. Peng. A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(06):823–855, Sept. 2009.

[6] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *ECOOP '93 - Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 406–431. Springer Berlin Heidelberg, July 1993. DOI: 10.1007/3-540-47910-4_21.

[8] D. E. Goldberg and J. H. Holland. Genetic Algorithms and Machine Learning. *Machine Learning*, 3(2-3):95–99, Oct. 1988.

[9] 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*, ICSE '95, pages 186–195, New York, NY, USA, 1995. ACM.

[10] J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. In *Working IEEE/IFIP Conf. on Software Architecture*, pages 12–21. IEEE, 2007.

[11] R. Krikhaar. *Software Architecture Reconstruction*. PhD dissertation, Univ. of Amsterdam, 1998.

[12] G. Lucassen, J. M. E. M. van der Werf, and S. Brinkkemper. Alignment of software product management and software architecture with discussion models. In *8th IEEE International Workshop on Software Product Management, IWSPM 2014, Karlskrona, Sweden, August 26, 2014*, pages 21–30. IEEE Computer Society, 2014.

[13] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[14] J. Peters, J. M. E. M. van der Werf, and J. Hage. Architectural pattern definition for semantically rich modular architectures. In *WICSA-Young Researchers*. IEEE, 2016.

[15] J. G. T. Peters. An Approach to Discovering Architectural Patterns in Software, 2016. MSc thesis.

[16] M. Pinzger and H. Gall. Pattern-supported architecture recovery. In *10th International Workshop on Program Comprehension, 2002. Proceedings*, pages 53–61, 2002.

[17] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.

[18] 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 WICSA 2014 Companion Volume, Sydney, NSW, Australia, April 7-11, 2014*, pages 8:1–8:8. ACM, 2014.

[19] L. Pruijt, C. Köppe, and S. Brinkkemper. Architecture compliance checking of semantically rich modular architectures: A comparative study of tool support. In *IEEE International Conference on Software Maintenance*, pages 220–229. IEEE, 2013.

[20] L. J. Pruijt, C. Köppe, J. M. E. M. 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, pages 851–854, New York, NY, USA, 2014. ACM.

[21] N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2011.

[22] K. Sartipi. Software architecture recovery based on pattern matching. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings*, pages 293–296, Sept. 2003.

[23] M. Sokolova, N. Japkowicz, and S. Szpakowicz. Beyond Accuracy, F-Score and ROC: A Family of Discriminant Measures for Performance Evaluation. In *AI 2006: Advances in Artificial Intelligence*, number 4304 in Lecture Notes in Computer Science, pages 1015–1021. Springer Berlin Heidelberg, Dec. 2006. DOI: 10.1007/11941439_114.

[24] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2010.

[25] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha. Recommending refactorings to reverse software architecture erosion. In *European Conference on Software Maintenance and Reengineering*, pages 335–340. IEEE, 2012.

[26] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design Pattern Detection Using Similarity Scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, Nov. 2006.

[27] E. J. Weyuker. Evaluation techniques for improving the quality of very large software systems in a cost-effective way. *Journal of Systems and Software*, 47(2-3):97–103, 1999.

[28] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. DiscoTect: A System for Discovering Architectures from Running Systems. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 470–479, Washington, DC, USA, 2004. IEEE Computer Society.