Scaling up the Bidirectional Maintenance Lifecycle

Gustavo Villavicencio

December 16, 2015

Abstract

The bidirectional maintenance lifecycle is based on two referential contexts (execution and maintenance), each characterized by a set of proper software attributes, and connected by a bidirectional refactoring mechanism. Such mechanism transforms the implementation version with the corresponding set of software attributes in the execution context to the version with the corresponding set of software attributes in the maintenance context, and in reverse. This paper proposes the integration of the design level in this model. The result of such integration should be a full-automatic environment for reengineering and maintenance with the capacity for solving maintenance requests at a specific abstraction level while keeping the structure and quality synchronizations with the other levels. In turn, such synchronizations should be adjusted to each referential context, i.e. execution and maintenance.

1 Introduction

The *bidirectional maintenance lifecycle* (BML) proposed in [27] is based on the idea that only one source code version is not enough for supporting two very different activities like *execution* and *maintenance*. Such differences are exposed by the software attributes 'demanded' by these contexts, that is, the execution context requires performance, security, energy efficiency, etc.; while the maintenance context requires comprehension, maintainability, extensibility, reusability, etc. In turn, the set of critical properties associated to a context is irrelevant for the other, for instance, performance and security are irrelevant when the system is in maintenance.

Connecting both referential contexts there are bidirectional refactoring sequences that transform the source code from the execution referential context (ERC) to the maintenance referential context (MRC), and in reverse. *Search Based Software Engineering* (SBSE) technique [8] is used to construct

such bidirectional refactoring sequences. The *metaheuristic algorithms* applied should be guided by a *fitness function* defined on the metrics associated to the set of software attributes 'relative' to each referential context. In this way, we maximize the software quality in both referential contexts avoiding the trade-off analysis between antagonistic sets of software properties. Basically, in order to solve a maintenance request, the maintainability conditions are improved by improving the relative set of properties in the MRC. To reach this target the system generates a sequence of intermediate (subsidiary) artifact versions, each one representing an improvement step of the metrics specified in the fitness function, and associated to the set of properties in the MRC. According to the maintenance request characteristics and his *cognitive style*, the maintainer solves the maintenance request in the most proper version. Then, the introduced modifications are automatically propagated to the version in the ERC by forward refactoring sequences, and so, the corresponding relative set of software properties is restored.

The forward refactoring sequence is obtained by reversing the reverse refactoring sequence that generated the subsidiary version where the modifications were introduced. The modifications can affect the execution of the forward refactoring sequence in different degrees: modifications do not influence the execution of the forward refactoring sequence and the whole sequence can be executed; only one part of the forward refactoring sequence can be executed since modifications have broken it at some point; the complete forward refactoring sequence can no longer be executed. Such degrees are related to the level at which the current version is reused in the new version. If the whole forward refactoring sequence can be executed, the whole (or almost the whole) current version has been reused in the new version generated after modifications. If only a subsequence of the forward refactoring sequence can be executed, then the current version has been partially reused in the new version generated after modifications. Finally, if the complete forward refactoring sequence can not longer be executed, nothing (or almost nothing) from the current version has been reused in the new version. If the latter case occurs, the reverse refactoring sequence should be discarded and the metaheuristic algorithm should calculate a new forward refactoring sequence from scratch. To do that, the system should 'reverse' (reset) the fitness function on the basis of the essential context attributes in the ERC. In this case, we say that the system *diverges* to an alternative solution.

We summarize the rationale and foundation for this model as follows:

- The increasing software complexity is being supported by only one ar-

tifact version which is used for both so dissimilar activities as execution and maintenance. To *decompress* the pressure on one artifact version only, we suggest to use alternative artifact versions grouped in two referential contexts, execution and maintenance, each characterized by a set of proper software properties.

- Regardless of the advancements in new techniques, tools, etc., for software maintenance, the maintenance costs continue growing. May be it is time software maintenance leaves the relaxed position adopted from the beginning, i.e. to execute their activities on exactly the same artifact version that is being executed. Here we are adopting a more proactive position by using subsidiary versions where the essential attributes for maintenance are maximized.

- The growing automation incorporated in the software refactoring process through the SBSE technique is a trend that will continue in the future. We are visioning that such trend will bring the necessary knowledge for constructing efficient refactoring sequences.

- The automatic propagation of changes between versions generated by refactoring is another element that gives meaning to this model. The modifications introduced during the maintenance phase into the subsidiary software versions can be propagated automatically to the version in execution [26]. In fact, the propagation of modifications from one representation to another has raised a new research area, i.e. *bidirectional transformations* (Bx) [5].

We consider that these facts provide well-founded evidence regarding the necessity and feasibility for the BML model. However, for the BML to be much more complete, the design level can be combined with the implementation level [1]. As a consequence, the extended version of the BML will combine two essential dimensions of a software artifact: *structure* and *quality*. The (vertical) structure axis is represented by the design and implementation levels, while the (horizontal) quality axis is represented by the sets of relative software properties in both referential contexts. The improved BML model we are visioning is depicted in figure 1. We consider that the design level incorporation is also well-founded since:

- SBSE technique can also operate on design representations [21, 13].

[1] An architecture level can also be integrated in this model but its treatment here will not enrich much more this paper.

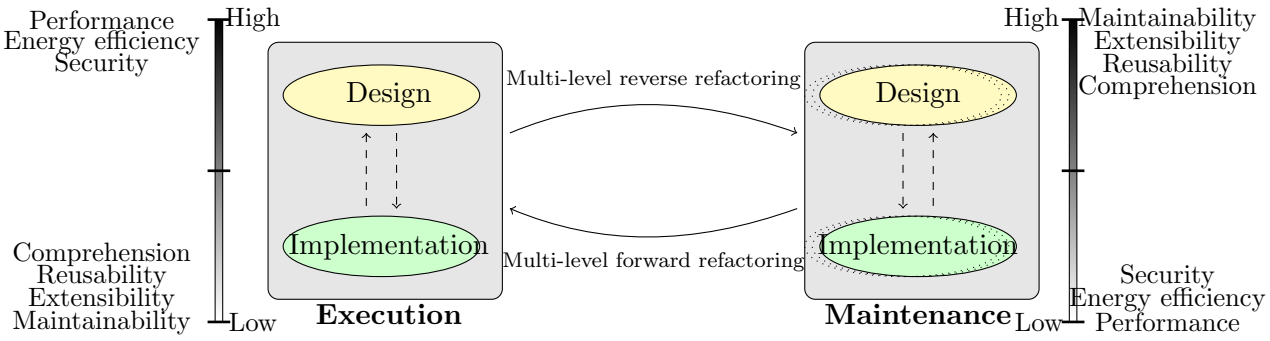

Figure 1: Integration of the design level into the bidirectional maintenance lifecycle from [27].

- Modifications can also be propagated between representations at this level.
- The synchronization between design and implementation representations is feasible and it is an active research area. Moreover, it is related to the Bx area.

As a result of such integration we draw a full-automatic environment for software reengineering and maintenance which, among others, should have the following benefits:

- It maximizes the design and implementation quality in both referential contexts (ERC and MRC) while keeping the quality and structure synchronizations in each of them.
- It should supply multi-perspective views from both levels, implementation and design, and at the same time shows how they should be related. Certainly this capacity should be valuable for the software maintenance team since it can be composed by programmers and designers who adhere to multiple *cognitive styles*.
- In the MRC the conditions are full improved since not only is the current representation of interest (design or implementation) improved by improving the corresponding attributes, but also the structure synchronization provides the *traceability* between different abstraction levels, which is essential for comprehension.

- It supplies a referential framework for organizing and classifying refactoring techniques. A refactoring (or refactoring sequence) is usually proposed for improving a software attribute but without considering the effects on the other attributes.

- The model makes one of the most complex task in the SBSE technique, that is, the fitness function formulation is easier since the set of metrics should be reduced according to the direction of refactoring.

The environment should be able to support a maintenance request at the design level and also keep the synchronization with the implementation level after modifications. Orthogonally, the environment should adjust the quality of each abstraction level in both referential contexts. Thus, the new BML model is based on two synchronization mechanisms operating in orthogonal ways: quality and structure synchronizations. The rest of the paper is organized as follows: In section 2 we describe the synchronization mechanims and how they should be integrated. In section 3 we discuss some issues raised from the strategy in the previous section, and finally section 4 shows the conclusions.

2 Scaling up the BML

According to Figure 1 quality and structural synchronizations are reached in both referential contexts, i.e., MRC and ERC. The following subsections describe how these processes should be.

2.1 Quality Synchronization

The separation between abstraction levels means that two different supports are used for both levels. Thus, in the OO paradigm, for instance, we would use UML diagrams for representing design while at the implementation level we would use C++ or Java languages. The support difference becomes evident in aspects like precision, completeness, abstraction, etc. It could be obvious, but such distinction is an essential issue since it states the metrics to be applied. In this way, from this perspective, we have three different metric categories:

- Metrics applied at the design level.

- Metrics applied at the implementation level.

- Metrics applied at both levels.

For instance, the DIT metric can be extracted and applied at both levels. The component balance metric, however, can only be applied at the implementation level but not at the design level. Such distinction is important for the quality synchronization process since we require a *correlation* or *consistency relationship* between attributes at both levels. That is, every quality variation on an abstraction level should be associated to a correlated or consistent quality variation on the other abstraction level. Note that we are not arguing that both variations are the same but they are related in a way that must be analyzed. In this regard, [11] states

> *It is further possible that there may be relationships in attributes across the different classes of metrics. For instance, the attributes number_of_lines which falls within the code group may be related to the branch_count and condition_count attributes in the design metric group.*

Thus, an *attribute correlation analysis* should set up a consistency relationship that should be the main element in the process of measuring the quality variations at both levels. At first sight, only the metrics in the third category should be applied in the consistency relation formulation, but a thorough correlation analysis should detect how the other metric categories correlate. In section 3 we will introduce more details on the quality correlation relationship. In section 2.3, we will use the symbol ($\sim$) to indicate that a specific design is qualitatively consistent with a specific implementation and in reverse.

On the other hand, SBSE technique has been applied successfully at both abstraction levels, design [21, 6] and implementation [17], to adjust the required quality. However, as we will point out in the following section, a consistency relationship between transformations at both abstraction levels is needed. That means that while the SBSE technique is applied on an abstraction level for constructing the refactoring sequence that adjust the quality, the refactoring sequence to be applied to the other abstraction level, is constructed from the previous one. In another way the synchronization will not be completed.

2.2 Structural Synchronization

Unlike quality synchronization, structure synchronization has received much more attention. One of the software engineering premise, though not always

fulfilled, has been to keep the consistency (synchronization) between the abstraction levels, i.e. design and implementation. This problem is usually called the *architectural conformance* [4]. The aim of an architectural conformance process is to detect the violations present in the source code that do not respect the constraints imposed by the intended architecture. Regarding this problem [24] states:

> *However, the effects embedded in the code cannot be automatically detected, because there is no language-level traceability between architectural design and its implementation. This traceability should be bidirectional, that is, a change in the code should be reflected in the corresponding design model. Unfortunately, current MDD tools are insufficient to realize this kind of bidirectional traceability.*

Thus, the problem is not always unidirectional throughout the software development (from design level to code level), but also bidirectional during the maintenance lifecycle (usually from code level to design level). Although, the latter is of our particular interest, we will see that the relationship from the design to the code is also essential in our context.

Much research effort has been devoted to keeping the consistency between the abstraction levels of a software artifact. Regarding the solutions, they have been focused from different perspectives: from the Architecture Description Languages (ADL) [1, 20, 24], reflexion models [18], domain-specific languages (DSL) [9], intensional views [15], design tests [3]. Furthermore, in the last years, a step forward has been taken in formalizing the synchronization between a design model and its implementation [14]. In this approach the object models are expressed in Alloy and the implementation in Java. In Figure 2 [2], Alloy laws (❶ and ❸) are applied on the object model and each of them has associated a synchronizer (❷ and ❹ respectively) which is applied to the implementation. The synchronizers, which are really program refinements, are composed and applied sequentially in order to obtain a synchronized implementation. Although in this work the synchronizers are unidirectional (from model to implementation) they can also be defined as bidirectional.

We should note that the association between the Alloy laws and synchronizers is an essential element in the synchronization process. [16] calls

[2] Note that a single transformation at the design level can entail a composition of many single transformations at the implementation level.

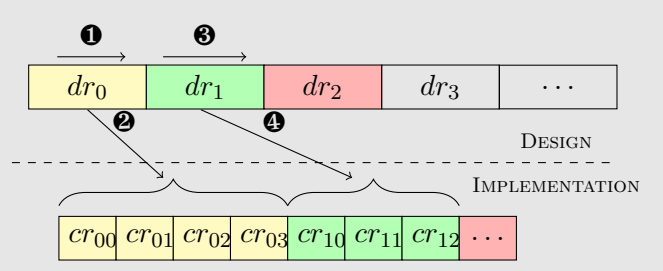

Figure 2: *M*apping the design refactorings (dr) to code refactoring (cr) using synchronizers.

reification a similar relationship between design-level refactorings and code-level refactorings. Thus, just as in the quality dimension, in the structure dimension we also have a correlation relationship between transformations at different abstraction levels. In section 2.3 we overload the ($\sim$) symbol to represent that a design level transformation is consistent with a code level transformation.

2.3 Integrating both Synchronization Mechanisms

In the two previous subsections we have presented the main mechanisms operating in horizontal (quality) and vertical (structure) ways. In both dimensions, the essential element is the correlation relationship, between metrics in the quality dimension, and between transformations in the structure dimension. We sustain that the integration of these elements will lead us to the construction of a more sophisticated synchronization mechanism useful for maintenance and reengineering.

At this point, and once notions of quality and structure synchronizations have been described, we can present the hypothesis that guides this work. After that, we will describe how such mechanisms should work when integrated. Let D and C be the design and implementation representations respectively, which are structural and qualitatively synchronized. Let also t_d and t_i be transformations to be applied at the design and implementation levels respectively and correlated between them (in terms of [14] t_d is an Alloy law and t_i its associated synchronizer). Finally, let a (polymorphic) function $\mathcal{Q}$ that measures the quality levels of both representations, D and C. Remember that the metrics at both abstraction levels also correlate. Our hypothesis states that, if t_d is applied on D to generate a new design representation D', and t_i is applied on C to generate a new implementation

representation C', then the quality measured by $\mathcal{Q}$ on D' correlates with the quality measures by $\mathcal{Q}$ on C'. More formally, and overloading the correlation relationship ($\sim$) on quality and structure

$$t_d \sim t_i \wedge \mathcal{Q}(D) \sim \mathcal{Q}(C) \Rightarrow \mathcal{Q}(t_d(D)) \sim \mathcal{Q}(t_i(C))$$

This hypothesis should have important consequences in the extended BML model due to its bidirectional nature. In the sequel we describe how the functioning of the maintenance and reengineering environment constructed on the BML model, and operating under the verification of the previous hypothesis should be. For this purpose, to the definitions already introduced, we added the following:

- D_0 and C_0 are the design and implementation versions in the ERC.
- t_d is decomposed in $\overleftarrow{dr}$ and $\overrightarrow{dr}$, the reverse and forward design-level refactoring sequences respectively, while t_i is decomposed in $\overleftarrow{cr}$ and $\overrightarrow{cr}$, the reverse and forward implementation-level refactoring sequences respectively.

Assuming that D_0 and C_0 are structurally and qualitatively synchronized in ERC, in the MRC the synchronization can be exposed at both levels as follows:

- Starting from D_0 in ERC, we say that if $D_i = \overrightarrow{dr}(D_0)$ synchronizes with $C_h = \overrightarrow{cr}(C_0)$ then $\mathcal{Q}(C_h) \sim \mathcal{Q}(D_i)$.
- Starting from C_0 in ERC, we say that if $C_h = \overrightarrow{cr}(C_0)$ synchronizes with $D_i = \overrightarrow{dr}(D_0)$ then $\mathcal{Q}(D_i) \sim \mathcal{Q}(C_h)$.

In the first case, the reverse refactoring sequence $\overrightarrow{dr}$ is obtained by SBSE technique while the related reverse refactoring sequence $\overrightarrow{cr}$ applied on the implementation level is obtained from the previous one through the correlation relationship. On the contrary, in the second case the reverse refactoring sequence $\overrightarrow{cr}$ is generated by SBSE technique, and $\overrightarrow{dr}$ by the correlation relationship. In the same way, according to what abstraction level the maintenance request was solved, the synchronization in the ERC can be exposed at both levels:

- Once the maintenance request has been solved on D_i, if $D_0 = \overleftarrow{dr}(D_i)$ synchronizes with $C_0 = \overleftarrow{cr}(C_h)$ then $\mathcal{Q}(C_0) \sim \mathcal{Q}(D_0)$. In this case $\overleftarrow{cr}$ should be generated by the correlation relationship.

- Once the maintenance request has been solved on C_i, if $C_0 = \overleftarrow{cr}(C_h)$ synchronizes with $D_0 = \overleftarrow{dr}(D_i)$ then $\mathcal{Q}(D_0) \sim \mathcal{Q}(C_0)$. In this case $\overleftarrow{dr}$ should be generated by the correlation relationship.

The last two definitions hold whenever $\overleftarrow{dr}$ and $\overleftarrow{cr}$ are obtained by (totally or partially) reversing $\overrightarrow{dr}$ and $\overrightarrow{cr}$ respectively. Clearly, there is an abuse in the previous notation since the variation of length of $\overleftarrow{dr}$ and $\overleftarrow{cr}$ is not represented. Such variation depends on which position the reverse sequence has been affected by the introduced modifications to solve the maintenance request. Furthermore, when the inversion fails, a new forward refactoring sequence should be calculated using SBSE technique: $\overleftarrow{dr'}$ or $\overleftarrow{cr'}$. In such cases, we say the system diverges to a new solution: D'_0 or C'_0 with new quality values. That is,

- Once the maintenance request has been solved on D_i, and the inversion of $\overrightarrow{dr}$ fails, if the new solution $D'_0 = \overleftarrow{dr'}(D_i)$ synchronizes with $C'_0 = \overleftarrow{cr'}(C_h)$ then $\mathcal{Q}(C'_0) \sim \mathcal{Q}(D'_0)$. In this case, $\overleftarrow{dr'}$ should be calculated by SBSE technique while $\overleftarrow{cr'}$ should be generated by means of the correlation relationship.

- Once the maintenance request has been solved on C_i, and the inversion of $\overrightarrow{cr}$ fails, if the new solution $C'_0 = \overleftarrow{cr'}(C_h)$ synchronizes with $D'_0 = \overleftarrow{dr'}(D_i)$ then $\mathcal{Q}(D'_0) \sim \mathcal{Q}(C'_0)$. In this case, $\overleftarrow{cr'}$ should be calculated by SBSE technique while $\overleftarrow{dr'}$ should be generated by means of the correlation relationship.

The introduction of modifications in order to solve a maintenance request represents a key moment in the process since the quality synchronization is at risk. It is a human-centered task. Once the modifications have been introduced either into the design or the code base, the quality should be checked to detect variations regarding the previous quality level. If the quality engineer detects unacceptable variations, the modifications should be adjusted until they reach the desired threshold. Just then the modifications should be propagated to the version in the ERC. Note that the structural synchronization in the MRC seems unnecessary after modifications, but it could be useful for testing. On the contrary, structural synchronization is essential previous modifications due to their usefulness for comprehension.

3 Discussion

One of the first issues raised in this approach is how to identify the most proper version for implementing the solution that solves a maintenance request. How to identify the initial set of components affected by a maintenance request has already be treated [2], but in this approach, such set of components (and their relationships) can be viewed from different perspectives. Thus, what perspective best fits the current maintenance request is the problem to solve. But the solution to this problem is strongly related to the programmer's cognitive style. Here a visual tool should be needed to quickly attract the programmer's attention on the code version that best fits his cognitive style. The integration of visualization capacities with refactoring tools has been suggested in [19]. [7] introduces the 'semantic lens' concept as a way to use the fitness function for tailoring the *pretty printer* to the cognitive style of the programmer. Furthermore, different types of 'cosmetic' modifications to the source code should be automatized in order to fit the code presentation to a programmer with a specific cognitive style [22]. Since modifications can also be carried out by designers in the design level, a similar rationale can be applied at this level. Therefore, an exhaustive experimentation in this area is needed to identify the specific characteristics of the visualizations required by the strategy. In this regard, [25] states

> *Although there are several visualization elements available, it is not a simple task to choose a visualization that will meet all the expectations and represent everything needed. Since the number of visualization alternatives keeps growing, it is important to adopt some sort of mechanism for selecting the most suitable ones, i.e., making an appropriate choice. In this sense, by selecting only the necessary features, visualizations can be composed with less effort.*

This work carries out a domain analysis on the visualization and interaction area, and organizes the information collected in a *visualization feature model*. Then, the model supports domain engineering activities for building new visualizations according to awareness requirements.

On the other hand, according to where the modifications are applied to solve a maintenance request, the relationship between design and code can be represented by Figure 3. That is, a minor movement on the big gear on the left has a great impact on the small gear on the right. And in reverse, a big movement on the small gear on the right has a slight impact on the

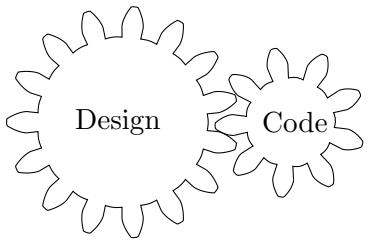

Figure 3: Visualizing the effects of changes in one abstraction level on the other level.

big gear on the left. That is, the modifications have different impacts according to the level they are applied to. If the modifications occur at the design level, the implementation could perceive a great scale impact. On the contrary, if the modifications occur at the code level they could not be perceived at the design level. From this observation, we pinpoint that the introduced modifications have two characteristics, i.e. *scope* and *intensity*. The latter one is particularly interesting in the current context. By intensity, we refer to the variations in the quality parameters as a consequence of the executed modifications at a specific abstraction level. According to the intuition described above, when these modifications are reflected by the other abstraction level, the intensity can be amplified or diminished depending on which level the modifications come from. For instance, if they are propagated from the design, they will have a great impact on the implementation, which should be reflected by important variations on the related metrics. Therefore, the modifications introduced into a level (design or implementation) have a specific impact on the attributes measured for such level, but the same measurement at the other level can deliver a different result, once the modifications have been reflected on it. The intensity with which the same modification is perceived by the abstraction levels, plays a central role in the quality correlation relationship formulation. The quality correlation relationship is the main mechanism used to synchronize the quality at both abstraction levels. Therefore, the correlation analysis mentioned in section 2.1 should end with a well-defined correlation relationship definition.

Another issue worth mentioning is the automatic generation of names throughout the transformation process. At either level, design and implementation, the search-based transformation entails the generation of intermediate representations before getting the representation with the desired quality level. Intermediate and final representations are constructed by elements (like functional units) whose names are assigned automatically. As

we know, such names are important for comprehension during maintenance. The automatic generation of meaningful names is an increasing issue in search-based transformation since its use is becoming more and more common. To solve this problem [10] proposes the construction of a dictionary based on the analysis of requirement documentation. In our context, this problem is also closely related with the structure synchronization since the assignment of names to related elements at both abstraction levels should have a unified criterion.

4 Conclusions

In this paper we have described the integration of the design level into the BML model, and we have shown that it is perfectly coherent and feasible. Regarding feasibility, the extended BML brings new challenges up, many of which have been identified, and also, the related bibliographic references provided. These references show the (dissimilar) progress made in each area. The fact that the challenges raised are being treated by the current software engineering research, at least in an incipient way, it is a good indicator for coherence and feasibility since the proposed model is a view on how future knowledge on the related areas could be integrated.

Even though the sophistication and complexity behind the change of perception on software maintenance and reengineering proposed here is important, they will be doubly rewarded, i.e. the most suitable version in both referential contexts. Such appropriateness is not only required in maintenance, as it was dealt with in section 1, but also during execution. In fact, there is a specific software engineering research area dedicated to improving the software properties for execution, i.e. *software performance engineering* [23]. What is more, other research areas ask for new properties during execution, such as the *green software* area that requires *efficiency energy consumption* [12] (Figure 1). Thus, the growing pressure put on only one artifact version for two so dissimilar activities will not be supported at large.

It is interesting to note that green software and software performance engineering areas, one as much as the other, use refactoring techniques for improving properties for execution. Even more interesting, just as in the software maintenance area, in both previous areas, refactorings are guided by the properties of interest without considering the others. Thus, in software maintenance, for instance, refactoring techniques are used for improving maintainability but without considering the consequence on performance. On the contrary, software performance engineering uses refactoring tech-

niques for improving performance without considering the effects on maintainability. As a consequence, there are software engineering research areas applying the same technique but in opposite directions. From this perspective, the BML model supplies a *general framework* in which the usefulness of the refactoring technique should be better understood.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 187–197, New York, NY, USA, 2002. ACM.

[2] G. Antoniol, G. Canfora, G. Casazza, and A. D. Lucia. Identifying the starting impact set of a maintenance request: A case study. In *CSMR*, pages 227–230, 2000.

[3] J. Brunet, D. S. Guerrero, and J. C. A. de Figueiredo. Structural conformance checking with design tests: An evaluation of usability and calability. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, pages 143–152, 2011.

[4] A. Caracciolo, M. F. Lungu, and O. Nierstrasz. A unified approach to architecture conformance checking. In *12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015, Montreal, QC, Canada, May 4-8, 2015*, pages 41–50, 2015.

[5] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective—GRACE meeting notes, state of the art, and outlook. In *ICMT2009 - International Conference on Model Transformation, Proceedings*, volume 5563 of *LNCS*. Springer, 2009.

[6] J. Denil, M. Jukss, C. Verbrugge, and H. Vangheluwe. Search-based model optimization using model transformations. In *System Analysis and Modeling: Models and Reusability - 8th International Conference, SAM 2014, Valencia, Spain, September 29-30, 2014. Proceedings*, volume 8769 of *Lecture Notes in Computer Science*, pages 80–95. Springer, 2014.

[7] M. Harman. Search based software engineering for program comprehension. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, pages 3–13, Washington, DC, USA, 2007. IEEE Computer Society.

[8] M. Harman and B. F. Jones. Search-based software engineering. *Information and Software Technology*, 43:833–839, 2001.

[9] D. Hou and H. J. Hoover. Using scl to specify and check design intent in source code. *IEEE Transactions Softw. Eng.*, 32(6):404–423, June 2006.

[10] A. C. Jensen and B. H. Cheng. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 1341–1348, New York, NY, USA, 2010. ACM.

[11] Y. Jiang, B. Cukic, T. Menzies, and N. Bartlow. Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, PROMISE '08, pages 11–18, New York, NY, USA, 2008. ACM.

[12] T. Johann, M. Dick, S. Naumann, and E. Kern. How to measure energy-efficiency of software: Metrics and measurement results. In *Proceedings of the First International Workshop on Green and Sustainable Software*, GREENS '12, pages 51–54, Piscataway, NJ, USA, 2012. IEEE Press.

[13] U. Mansoor, M. Kessentini, P. Langer, and T. Mayerhofer. Multi-view model refactoring using a multi-objective evolutionary algorithm. *Software Quality Journal*.

[14] T. Massoni, R. Gheyi, and P. Borba. Synchronizing model and program refactoring. In *Proceedings of the 13th Brazilian Conference on Formal Methods: Foundations and Applications*, SBMF'10, pages 96–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views — a case study. *Journal on Computer Languages, Systems and Structures*, 32(2–3):140–156, July-October 2006. Special Issue: Smalltalk.

[16] I. H. Moghadam and M. Ó. Cinnéide. Automated refactoring using design differencing. In *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, pages 43–52, 2012.

[17] I. H. Moghadam and M. Ó Cinnéide. Code-imp: A tool for automated search-based refactoring. In *Proceedings of the 4th Workshop on Refactoring Tools*, WRT '11, pages 41–44, New York, NY, USA, 2011. ACM.

[18] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT '95, Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering, Washington, DC, USA, October 10-13, 1995*, pages 18–28, 1995.

[19] E. R. Murphy-Hill. Improving usability of refactoring tools. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, pages 746–747, Portland, Orgegon, USA, October 2006. ACM.

[20] L. G. P. Murta, A. van der Hoek, and C. M. L. Werner. Archtrace: Policy-based support for managing evolving architecture-to-implementation traceability links. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 135–144, 2006.

[21] O. RäIhä. Survey: A survey on search-based software design. *Computer Science Review*, 4(4):203–249, Nov. 2010.

[22] S. P. Reiss. Automatic code stylizing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 74–83, New York, NY, USA, 2007. ACM.

[23] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1990.

[24] N. Ubayashi, J. Nomura, and T. Tamai. Archface: A contract place where architectural design and code meet together. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 75–84, New York, NY, USA, 2010. ACM.

[25] R. Vasconcelos, M. Schots, and C. Werner. An information visualization feature model for supporting the selection of software visualizations. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 122–125, 2014.

[26] G. Villavicencio. A new software maintenance scenario based on refactoring techniques. In *16th European Conference on Software Maintenance and Reengineering (CSMR 2012)*, Zseged, Hungary, March 2012. IEEE.

[27] G. Villavicencio. Software maintenance like maintenance in other engineering disciplines. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 853–856, New York, NY, USA, 2014. ACM.