From Products to Product Lines Using Model Matching and Refactoring

Julia Rubin
University of Toronto, Canada
and
IBM Research at Haifa, Israel
e-mail: mjulia@il.ibm.com

Marsha Chechik
University of Toronto, Canada
e-mail: chechik@cs.toronto.edu

***Abstract*—In this paper, we suggest a method for refactoring UML structural and behavioral models of closely related individual products into product lines. We propose to analyze duplications in the models of individual products using a heterogeneous match algorithm which takes into account structural and behavioral information to identify identical and similar model elements. Identical elements (exact matches) are refactored to common parts of the product line, similar elements are refactored to variable alternative parts, and unmatched elements are refactored to variable optional parts. We further propose to adjust the quality of the match by analyzing quality of the resulting refactoring. We evaluate UML comprehensibility before and after the change using prediction models that are based on static metrics, and use the results to set the optimal thresholds for identity and similarity between model elements. We illustrate our proposed approach on an example.**

I. Introduction

A *software product line (SPL)* is a set of software-intensive products sharing a common, managed set of features that satisfy the specific needs of a particular market segment [1]. SPL *commonalities* represent artifacts that are part of each product of the product line, while SPL *variabilities* represent artifacts that are specific to one or more (but not all) individual products [2].

There are different approaches to SPLs implementation, such as aspect weaving, conditional compilation, code generation and more [2], [3], [4], [5], [6], all aiming to implement variability in SPLs in a maintainable way. In reality, SPLs often emerge from experiences in successfully addressed markets with similar, yet not identical needs. It is difficult to foresee these needs a priori and hence to design an SPL upfront. To address the different marketing needs, software developers often create new products ad-hoc, by using one or more of the available techniques such as duplications (the clone-and-own approach), inheritance, source control branching, componentization and more.

In a production environment, the use of the ad-hoc methods does not provide sufficient management. In many cases, even when a more mature reuse technique, such as inheritance or componentization [7], [8] is chosen, software artifacts are used by different products, each with its own release schedule. In such cases, duplication is the fastest possible solution often undertaken. For embedded and resource constrained environments, where deployed resources should be kept to a minimum, inheritance is not desired, as it requires fine-grained fragmentation to support feature-based modularization and results in an undesired executable blow up. Moreover, most of the reuse techniques are inapplicable to software development artifacts other than code, which, again, results in duplications of these artifacts. As the number of variants increases, the effort for maintaining them increases, too as changes in the shared artifacts must be repeated in all variants.

Modern SPL techniques, such as the approaches described by Pohl et al. [2] and Gomaa [3], support handling of a collection of similar software development artifacts through the entire development life-cycle and enable efficient variability management and reuse. In order to take advantage of these techniques, we need to refactor legacy product lines into the suggested representations. The first step towards obtaining a product line model with common and variable parts is comparison of existing products and determination of their commonalities and variabilities. We consider two separate cases of consolidation: where the variants are maintained independently and are being refactored into one product line model, and where refactoring is performed on a single model that uses suboptimal variability mechanisms. In the latter case, the goal is to transform the model into a product line with common and variable parts, while preserving its expressiveness.

In this paper, we propose to support refactoring by analyzing duplications in the models of individual products using a heterogeneous match algorithm which takes into account structural and behavioral information to identify identical and similar model elements. Identical elements map to *common* SPL artifacts, similar elements map to *variable alternative* SPL artifacts, and unmatched elements map to *variable optional* SPL artifacts.

We further propose to perform a *refactoring* of the original model(s) into a model with commonalities and variabilities, and to use static metrics of model comprehensibility to evaluate the quality of the resulting refactoring. The refactoring process is performed iteratively and interactively, identifying the optimal thresholds for identity and similarity between model elements, until a model of a desired quality is produced.

The remainder of this paper is organized as follows. Sec-

tion II describes related research. Section III introduces the motivating example. In Section IV, we outline the proposed heterogeneous match and refactoring algorithm, illustrating it on the motivating example in Section V. Section VI concludes the paper and discusses future work.

II. Background

Several UML match and merge algorithms [9], [10], [11], [12], [13], [14], [15], [16] exist. Of particular importance are those among them that are able to find not only fully identical but also approximate matches [9], [13], [14], [15]. However, all these algorithms are designed for a particular type of UML diagrams and do not take into account heterogeneous information obtained by analyzing several types of diagrams together. Both similarity detection algorithms and model-level clone-detection techniques for other model types, such as Matlab/Simulink models [17], [18], also consider a single system perspective. Code-level clone-detection techniques [19] are not applicable to models.

UML model refactoring [20], [21], [22], as well as code refactoring techniques [23], [24] largely focus on improving the internal structure of a software system. These techniques usually *add*, *move*, *remove*, *rename*, *generalize*, *specialize* or *modularize* software elements to improve software design, rather than focusing on identifying and managing its common and variable parts.

Several approaches aim at building product lines out of legacy artifacts, e.g., [25], [26], [27], [28]. These approaches mainly discuss guidelines, methodologies and lessons learned, without providing details on software artifact analysis and tool support. Code-level refactoring techniques that reconstruct product lines from legacy code systems are proposed in [29], [30], [31]. Kosche et. al [32] introduce a method to reconstruct the static architecture of variants using the reflection method. While the motivation of this work is close to ours, it focuses on the extension of the Murphy's reflection method from single systems to software variants and uses code clone detection techniques to identify matching variants.

Our work is different from the above as it operates on the level of UML models and considers more than a single UML diagram type to identify matching variants.

III. Motivating Example

Our work is motivated by a real-life UML model of a partner. Since the details of that model cannot be made public, we constructed a representative example that allows us to demonstrate the nature of the model without revealing any partner-specific details.

Figure 1 presents a UML2 class diagram depicting the main classes of a `Washing Machine` which has several `Wash` programs. `Gentle Wash` program is further separated into `Wool`, `Hand` and `Cold` washes. The `Wash` program controls `Spinner`, `Water Heater` and `Program Sequence` components of the washing machine. This washing machine can perform three wash sequences: `Regular`, `No Spin` and `Double Rinse`.

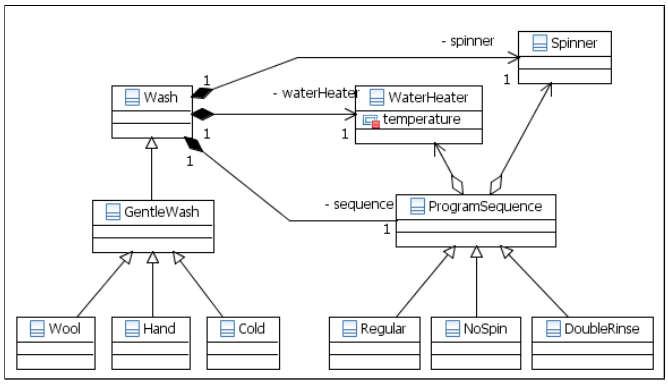


Fig. 1. Structure of the Washing Machine example.

Fine-grain separation into different wash programs and sequences allows the vendor to produce multiple washing machine models with different combination of features. For example, it can offer consumers a washing machine that performs the `Regular` wash sequence for the `Wool` and `Cold` programs, and the `No Spin` wash sequence for the `Hand` program. This configuration is depicted in Figure 2. The vendor can also choose to produce washing machines without the `Hand` wash program or those without the `Double Rinse` wash sequence.

Since the software of the washing machine is installed in a resource-constrained environment, a requirement is not to ship systems with extra software. That is, a washing machine model without the `Double Rinse` wash sequence should not have the code for that feature *physically* present. The modularity of the solution, however, results in a large set of duplications, since the components of the solution should be deployed and function independently.

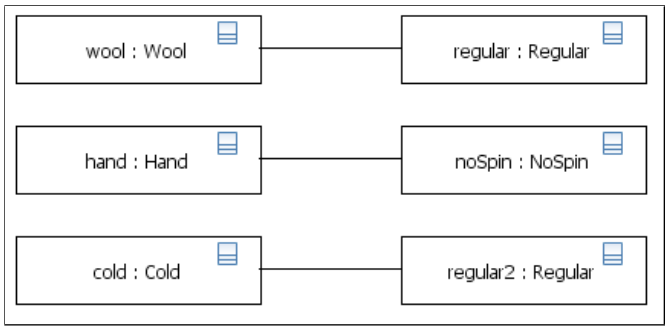
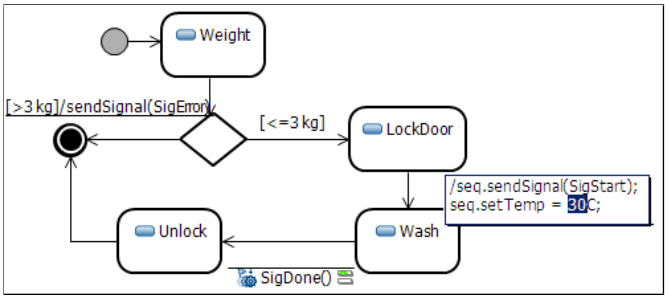


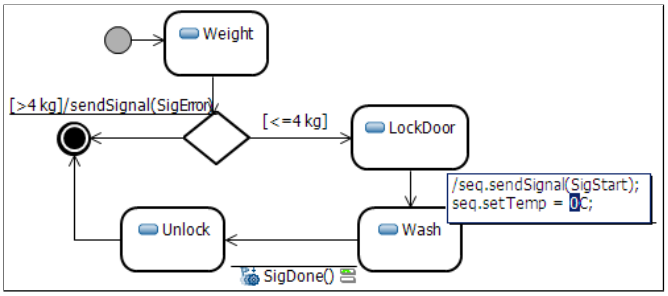
Fig. 2. A Model of the Washing Machine.

The behavior of the washing machine components is depicted in Figures 3-4. Figure 3(a) shows the behaviour of the `Wool` and the `Hand` washes, which are identical. Figure 3(b) shows the behaviour of the `Cold` wash, which differs from the `Wool` and the `Hand` washes in the set wash temperature. Figures 4(a), 4(b) and 4(c) show `Regular`, `No Spin` and `Double Rinse` program sequences, respectively. Wash programs communicate with wash sequences using signals. After a wash program statechart completes the `LockDoor` state, it sends out the `SigStart` signal. Wash sequence

statecharts wait for that signal and start to operate upon its reception. When the entire wash sequence is completed, the wash sequence statechart sends out a `SigDone` signal, which indicates that the wash program can proceed to the `Unlock` state.



(a) Behaviour of `Wool` and `Hand` Washes.



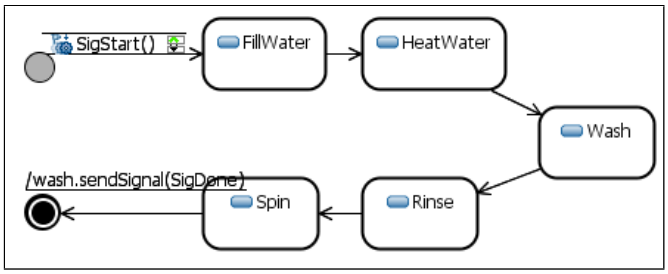
(b) Behaviour of `Cold` Wash.

Fig. 3. Wash Program Statecharts.

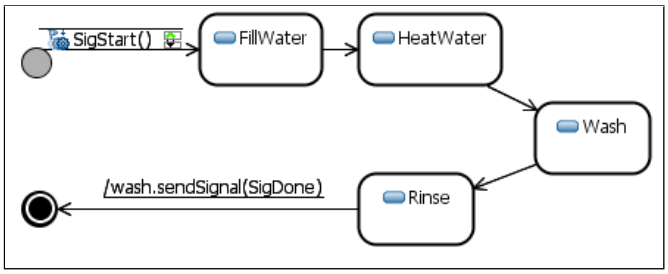
The washing machine model in the described example essentially represents a *product line* of control software for different washing machines. The software can be built by selectivity including and excluding some of the model's components. Our goal is to refactor the model by explicating its common and variable part, while eliminating duplication and reducing complexity. At the same time, our goal is to maintain modularity and minimal size of the deployed code.

IV. The Proposed Approach

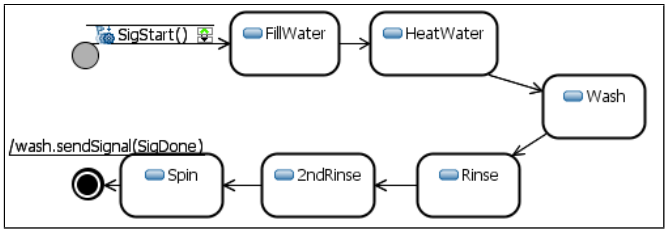
Our main objective is to define and implement the *3-Valued heterogeneous Match and Refactoring algorithm (ThreeVaMaR)*. The algorithm accepts as input a model with duplications which represent multiple product variants. The algorithm then produces as output the model of a product line, while explicating product line commonalities and variabilities, following the approach in [2], [3]. For the cases when the algorithm is used to merge two or more product variant models, the input model is constructed by computing the union of these variant models, without any merging. The algorithm works iteratively, by repeating the sequence of four basic steps: measuring the quality of the input model, finding duplications and near duplications via matching, refactoring and comparing the quality of the refactored model to that of the original one. After each iteration, the threshold of the match is decreased, which results in an increased number of matches. The algorithm stops when a model of the desired quality is produced. These basic steps are outlined in Algorithm 1 and further elaborated below.



(a) `Regular`.



(b) `No Spin`.



(c) `Double Rinse`.

Fig. 4. Wash Sequence Statecharts.

Algorithm 1 ThreeVaMaR(M: model)

1: $Q_{ref} \leftarrow$ *quality*(M)
2: $T_i \leftarrow$ initT_i //initial identity threshold
3: $T_s \leftarrow$ initT_s //initial similarity threshold
4: **repeat**
5: $Q_{orig} \leftarrow Q_{ref}$
6: $matches \leftarrow$ *match*(M, T_i, T_s)
7: $M_{ref} \leftarrow$ *refactor*(M, $matches$)
8: $Q_{ref} \leftarrow$ *quality*(M_{ref})
9: decrease T_i
10: decrease T_s
11: **until** $Q_{ref} > Q_{org}$

A. Evaluation of Model Quality

Lines 1 and 8 of the algorithm use static metrics proposed in the literature to assess comprehensibility of UML class diagrams (*quality*).

Specifically, the work in [33] suggests that an increased number of classes, inheritance and aggregation mechanisms results in an increased cognitive complexity of a UML class diagram. A quantitative assessment of the complexity is made by

two different prediction models: the first, which is proposed in [34] and [35], is based on *Fuzzy Deformable Prototypes* [36], [37]. This model allows predicting understandability time, and thus maintainability, of a diagram by calculating a triangular fuzzy number for each new diagram and assessing its affinity with predefined prototypes that were obtained from responses given by humans during controlled experiments. Each triangular fuzzy number represents the degrees of membership in three prototype groups – `easy`, `medium`, and `difficult`.

Another prediction model for UML class diagram understandability time is suggested in [38]. This model is based on the *Multivariate Linear Model* [39] which includes dependant and independent variables (metric values) in the prediction model, as long as they fulfil predefined statistical criteria.

The authors of [40] suggest that the number of states, actions and transitions of a statechart diagram correlates with its increased cognitive complexity. A prediction model that quantitatively confirms this finding, by assessing the understandability times, is proposed in [41]. The model is based on the *Individual Regression Equations* [42] – a technique similar to the *Multivariate Linear Model*.

We propose to use all three of the above-mentioned metrics: the first two for comparing class diagrams (see Table I), and the third one – for statechart diagrams (see Table II). It should be noted that low values of quality calculations (lines 1 and 8) are associated with high model understandability and vise-versa.

B. Match

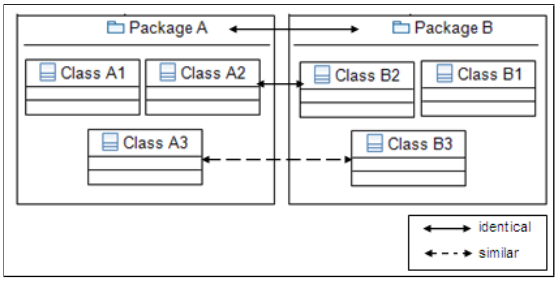
By examining the example in Section III, we can see that classes `Wool` and `Hand`, depicted in Figure 1, could be matched because they have the exact same behavior, shown in Figure 3(a). The behaviour of the class `Cold`, on the other hand, is not identical to that of the classes `Wool` and `Hand`. Thus, the similarity between `Cold` and the other two classes is lower. Yet such difference between the three classes cannot be detected if only the structural information, presented in the class diagrams, is considered: all three classes have the same static structure, the same set of methods (not shown) and the same set of relationships to other model elements.

This means that in order to determine the similarity of UML model elements, it is essential to consider the collective information captured in different UML diagrams and views: the heterogeneous match algorithm should weigh structural, behavioral and, if available, code snippet similarity when calculating the similarity degree between model elements. That is, the similarity between classes should take into account the similarity between their corresponding statecharts and/or activity diagrams; the similarity between states should take into account the similarity between their actions' code snippets, etc.

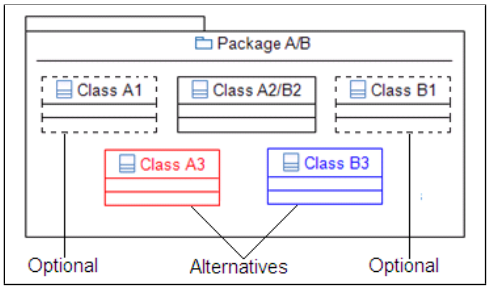
We are not aware of heterogeneous match algorithms proposed in the literature and thus intend to implement such an algorithm as part of our work (line 6 of Algorithm 1). Furthermore, since we intend to detect exact and near duplications in the input models for the purpose of refactoring those into models with commonalities and variabilities, Algorithm 1 defines two thresholds – *identity degree threshold* (T_i in line 2) which specifies elements that are considered identical for the purpose of the refactoring, and *similarity degree threshold* (T_s in line 3) which specifies elements that are considered similar. While many heuristic matching techniques define identity threshold experimentally, we suggest to adjust both thresholds dynamically during the refactoring process, by measuring quality of the resulting refactored model. As long as the complexity of the resulting model does not exceed the complexity of the model in the previous iteration, we proceed with the process of lowering the thresholds (lines 9-10). We stop when additional refactoring does not lead to an improvement in the model.

C. Refactoring

Model refactoring (line 7 of the Algorithm 1) is performed top-down, starting from the high-level model elements, such as packages or classes. It is built upon the idea that identical elements are transformed into product line *common* artifacts, which requires a recursive merge of their internal structure. When the internal structure of these elements is merged, identical, similar and unmatched elements are recursively handled: similar elements are transformed into product line *variable alternative* artifacts, unmatched elements are transformed into *variable optional* artifacts, while identical elements are merged and the process continues. For example, when two packages depicted in Figure 5(a) are identified as identical, their corresponding internal elements – classes, interfaces and sub-packages – are to be composed. In this case, both classes `A1` and `B1` are marked optional, classes `A3` and `B3` are marked as alternatives, while classes `A2` and `B2` are recursively merged. The result is shown in Figure 5(b).



(a) Before the Refactoring.



(b) After the Refactoring.

Fig. 5. Identity and Similarity Relationships.

V. Illustration

In this section, we illustrate the use of the *ThreeVaMaR* algorithm for the example described in Section III. Specifically, we instantiate the algorithm on the model in Figure 1, while manually simulating the match and refactoring capabilities. Below, we propose and analyze three possible refactorings of the example model.

A. Refactoring #1

Combining `Wool` and `Hand` classes together results in a model presented in Figure 6. Following [33], decreasing the number of classes, inheritance and aggregation mechanisms results in a decreased cognitive complexity; thus, the refactored model is slightly simpler than the original one.

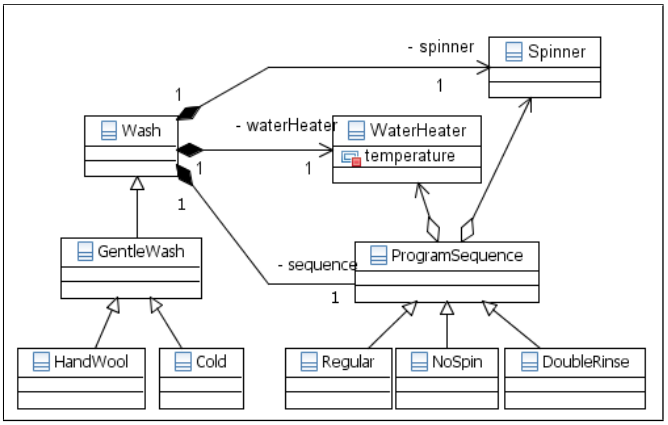


Fig. 6. Washing Machine Structure After Refactoring.

The quantitative complexity assessment of the original model in Figure 1 and the refactored model in Figure 6 is shown in Table I. In *Fuzzy Deformable Prototypes*-based prediction model, the original class diagram has 45% affinity with the `easy` case and 29% with the `medium` case, while the refactored class diagram has 48% affinity with the `easy` case and 26% with the `medium` case. The change is statistically insignificant because the changes between these two class diagrams are minor, but we are able to demonstrate stability of the metrics. The prediction model that is based on *Multivariate Linear Model* does not differentiate between the original and the refactored class diagrams at all.

Prediction Method	Original Model	Ref#1	Ref#2	Ref#3
Fuzzy Deformable Prototypes	0.292 (0.45,0.29,0)	0.285 (0.48,0.26,0)	0.163 (0.92,0,0)	0.089 (0.99,0,0)
Multivariate Linear Model	215.17	215.17	91.3	91.3

TABLE I
Predicted Understandability for Washing Machine Class Diagram.

While the complexity of each individual statechart in the refactored UML model did not change, the new washing machine has fewer statecharts, and thus its complexity is reduced.

In addition, in the refactored washing machine UML model there is no need to create a composite structure diagram for each combination of washing machine features that define a washing machine model (like the one in Figure 2). All configurations can be represented in one diagram and be controlled by a set of washing machine features (similarly to the method suggested in [2] and [3]). Figure 7 depicts such a diagram, where `Wool` and `Hand` features are attached to UML model elements. The `Wool` feature corresponds to the combination in which `HandWool` uses the `Regular Spin` sequence, as in the `Wool` wash of the model in Figure 2; the `Hand` feature corresponds to the combination in which `HandWash` uses the `No Spin` sequence, as in the `Hand` wash of the model in Figure 2.

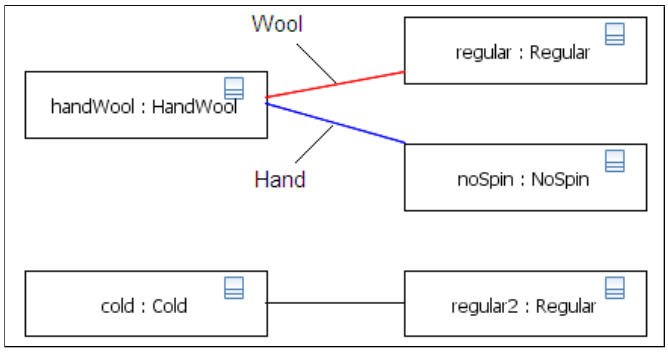


Fig. 7. A Washing Machine Model After Second Refactoring.

B. Refactoring #2

By decreasing the identity degree threshold, `Hand`, `Wool` and `Cold` classes can all be matched together. In addition, all three program sequence classes – `Regular`, `No Spin` and `Double Rinse` – are matched together as well. The class diagram of the refactored model is shown in Figure 8.

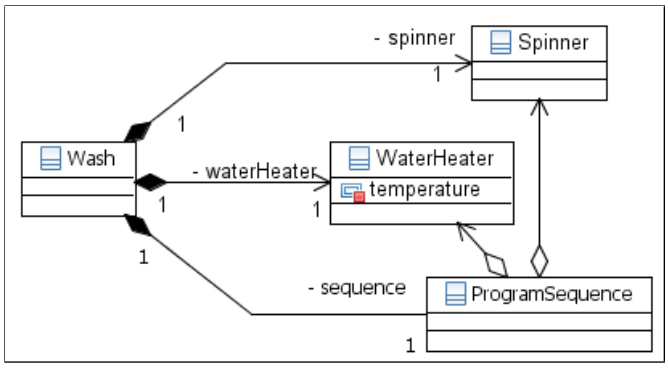


Fig. 8. Washing Machine Structure After Refactoring #2.

Statechart diagrams of the `Wash` program and `Program Sequence` are depicted in Figure 9 and 10, respectively. For each of these diagrams, elements that passed the identity degree threshold were merged; elements that passed the similarity degree threshold were defined as alternatives (e.g., transitions from the `Weight` state in Figure 9), and all

other elements were defined as optional (e.g., `2ndRinse` state in Figure 10). The diagrams are again controlled by features. `WoolHand` and `Cold` features control different wash programs, while `Regular`, `No Spin` and `Double Rinse` features control different wash sequences.

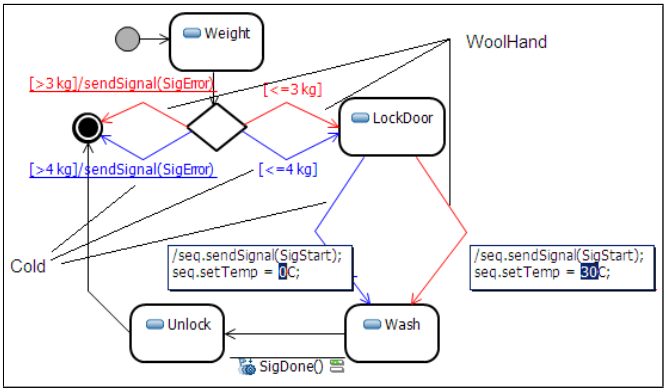


Fig. 9. Wash Program Statechart After Refactoring #2.

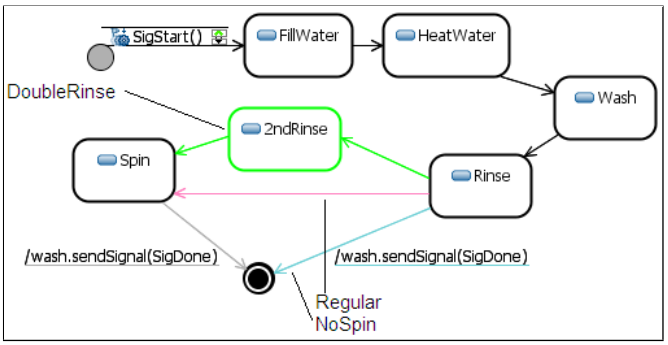


Fig. 10. Wash Sequence Statechart After Refactoring #2.

With respect to the complexity metrics, we obviously reduced the number of classes, inheritance and aggregation mechanisms, which, following [33], results in a decreased cognitive complexity. Calculated understandability measures shown in Table I confirm that the refactored class diagram is significantly simpler than both the original one and the one we obtained after the first refactoring iteration.

Following [40], the number of states, actions and transitions correlates with an increased cognitive complexity for a statechart diagram. Even though each refactored diagram has more transitions than the original one, we reduced the total number of statechart diagrams from five to just two, which leads to a decreased complexity. This is quantitatively confirmed using the prediction model in [41]. Since the metrics and the prediction model that we used do not take into account associations of state machine elements with features, and since in statecharts a guard on a transition is the closest semantic equivalent to the association of a transition with a feature, we adapted the metrics to count these associations as additional guards. The result of the calculations is summarized in Table II. As expected, the statechart diagrams after the second refactoring are easier to understand and thus to maintain.

C. Refactoring #3

By setting the identity degree threshold even lower than in Refactoring #2, `Hand`, `Wool`, `Cold`, as well as `Regular`, `No Spin` and `Double Rinse` classes can all be matched together. The class diagram of the refactored model is depicted in Figure 11.

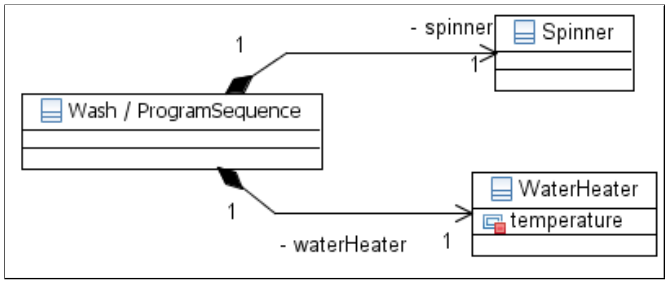


Fig. 11. Washing Machine Structure After Refactoring #3.

The statechart diagram of the new `Wash/Program Sequence` class is shown in Figure 12. This diagram is again controlled by features. `WoolHand` and `Cold` features control different wash programs, while `Regular`, `No Spin` and `Double Rinse` features control different wash sequences. Wash program statecharts (Figure 3) do not match with the wash sequence statecharts (Figure 4). Thus, they are placed in orthogonal regions of the refactored statechart diagram, with a `fork` pseudostate selecting which region to execute based on features (which are omitted for simplicity).

With respect to the metrics, while we further reduced the number of classes when comparing to Refactoring #2, the calculated understandability measures shown in Table I did not substantially change. Statechart-related calculations shown in Table II show a slight improvement in the statechart understandability time. However, the metrics that we used for this calculation were not designed to take into account orthogonal statechart regions as well as the `fork` and the `join` pseudostates. Thus, we attribute the observed improvement to the lack of appropriate measures, as opposite to a real complexity benefit. We conclude that the improvement gained by this additional refactoring is minor, if it exists at all, and thus the refactoring is not essential.

Overall, we conclude that the second refactoring was the most effective. The ability to specify different feature combinations and thus to easily create software for new washing machine models is an additional benefit of the resulting refactored model, comparing to the original one.

Prediction Method	Original Model	Ref#1	Ref#2	Ref#3
Individual Regression Equations	746.175	617.462	282.852	187.112

TABLE II
PREDICTED UNDERSTANDABILITY FOR WASHING MACHINE STATECHART DIAGRAMS.

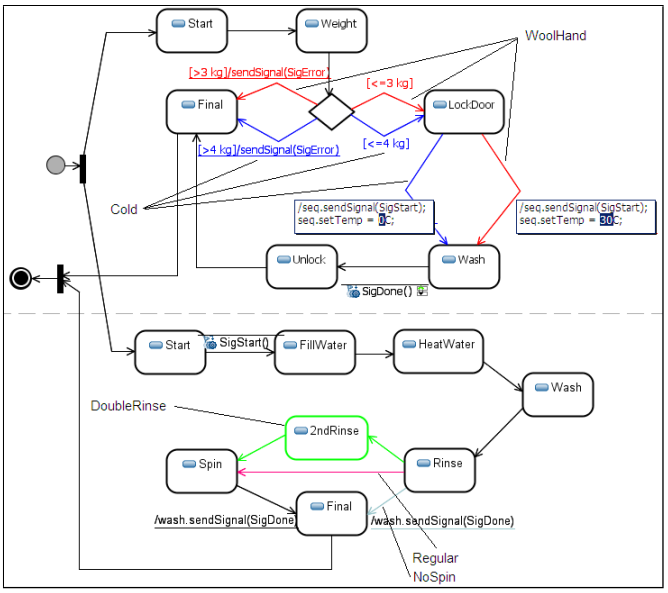


Fig. 12. Wash Program Statechart After Refactoring #3.

VI. Discussion and Future Work

In this paper, we outlined the proposed *heterogenous match and refactor algorithm (ThreeVaMar)*, which we intend to employ for refactoring individual product variants into product line models with common and variable parts. Our algorithm uses two thresholds: the *identity degree threshold* which specifies elements that are considered common in the product line, and the *similarity degree threshold* which specifies elements that are considered alternatives. We propose to perform the refactoring of individual product variants into product line models iteratively, by dynamically setting the thresholds that determine common and alternative elements, and proceed with the process of decreasing thresholds until a model of the desired quality is produced.

The quality of the match and subsequent refactoring in our proposed algorithm relies on being able to measure complexity of the resulting model. Part of our goals is to evaluate the feasibility of using the complexity prediction model that is based on static metrics of the UML model complexity in order to measure quality. The initial evaluation shows promising results. However, the stability and reliability of the prediction model are yet to be determined. We also plan to investigate the usage of alternative prediction models, or develop our own prediction model adjusted to the unique needs of the product line domain.

Our work aims at optimizing the input model by creating its refactoring of the lowest possible complexity. The suggested algorithms gradually decreases the identity and similarity thresholds, and stops when the complexity of the resulting refactoring stops deceasing or starts to grow. We still need to see whether alternative search techniques are more effective in finding the desired optimal refactoring.

Our next obvious step is to define a matching algorithm with the desired heterogeneous characteristics and the three-way outcome and build the iterative refactoring platform around it. We then intend to validate the effectiveness of our technique on real-life models from our industrial partner.

References

[1] P. C. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, ser. SEI Series in Software Engineering. Addison-Wesley, August 2001.

[2] K. Pohl, F. Guenter Boeckle, and van der Linden, *Software Product Line Engineering : Foundations, Principles, and Techniques*. New York, NY: Springer, 2005.

[3] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[4] Big Lever, Last Accessed: June 2010. [Online]. Available: http://www.biglever.com/solution/product.html

[5] pure::systems, Last Accessed: June 2010. [Online]. Available: http://www.pure-systems.com/

[6] M. Voelter and I. Groher, "Product Line Implementation using Aspect-Oriented and Model-Driven Software Development," in *Proceedings of the 11th International Software Product Line Conference (SPLC'07)*. IEEE Computer Society, 2007, pp. 233–242.

[7] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[8] C.-M. Park, S. Hong, K.-H. Son, and J. Kwon, "A Component Model Supporting Decomposition and Composition of Consumer Electronics Software Product Lines," in *Proceedings of the 11th International Software Product Line Conference (SPLC'07)*. IEEE Computer Society, 2007, pp. 181–192.

[9] Z. Xing and E. Stroulia, "UMLDiff: an Algorithm for Object-Oriented Design Differencing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*. ACM, 2005, pp. 54–65.

[10] U. Kelter and M. Schmidt, "Comparing State Machines," in *Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models (CVSM'08)*. ACM, 2008, pp. 1–6.

[11] U. Kelter, J. Wehren, and J. Niere, "A Generic Difference Algorithm for UML Models," in *Software Engineering*, ser. LNI, vol. 64. GI, 2005, pp. 105–116.

[12] A. Mehra, J. Grundy, and J. Hosking, "A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*. ACM, 2005, pp. 204–213.

[13] M. Girschick, "Difference Detection and Visualization in UML Class Diagrams," TU Darmstadt, Tech. Rep., 2006.

[14] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave, "Matching and Merging of Statecharts Specifications," in *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. IEEE Computer Society, 2007, pp. 54–64.

[15] K. Bogdanov and N. Walkinshaw, "Computing the Structural Difference between State-Based Models," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering (WCRE'09)*. IEEE Computer Society, 2009, pp. 177–186.

[16] H. Liu, Z. Ma, L. Zhang, and W. Shao, "Detecting Duplications in Sequence Diagrams Based on Suffix Trees," in *Proceedings of the XIII Asia Pacific Software Engineering Conference (APSEC'06)*. IEEE Computer Society, 2006, pp. 269–276.

[17] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert, "Clone Detection in Automotive Model-Based Development," in *Proceedings of the 30th International Conference On Software Engineering (ICSE'08)*. ACM, 2008, pp. 603–612.

[18] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Complete and Accurate Clone Detection in Graph-Based Models," in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, 2009, pp. 276–286.

[19] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: a Qualitative Approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.

[20] S. Hosseini and M. A. Azgomi, "UML Model Refactoring with Emphasis on Behavior Preservation," in *Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'08)*. IEEE Computer Society, 2008, pp. 125–128.

[21] G. Sunyé, D. Pollet, Y. L. Traon, and J.-M. Jézéquel, "Refactoring UML Models," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML'01)*. Springer-Verlag, 2001, pp. 134–148.

[22] M. Boger, T. Sturm, and P. Fragemann, "Refactoring Browser for UML," in *Proceedings of the International Conference on Objects, Components, Architectures, Services, and Applications for a Networked World (NODe '02)*. Springer-Verlag, 2003, pp. 366–377.

[23] T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[24] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[25] S. Ferber, J. Haag, and J. Savolainen, "Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line," in *Proceedings of the Second International Conference on Software Product Lines (SPLC'02)*. Springer-Verlag, 2002, pp. 235–256.

[26] J. Bayer, J.-F. Girard, M. Würthner, J.-M. DeBaud, and M. Apel, "Transitioning Legacy Assets to a Product Line Architecture," in *Proceedings of 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'99)*. Springer-Verlag, 1999, pp. 446–463.

[27] K. C. Kang, M. Kim, J. Lee, and B. Kim, "Feature-oriented Reengineering of Legacy Systems into Product Line Assets," in *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, 2005, p. 45.

[28] K. Kim, H. Kim, and W. Kim, "Building Software Product Line from the Legacy Systems: Experience in the Digital Audio and Video Domain," in *Proceedings of the 11th International Software Product Line Conference (SPLC'07)*. IEEE Computer Society, 2007, pp. 171–180.

[29] J. Liu, D. Batory, and C. Lengauer, "Feature Oriented Refactoring of Legacy Applications," in *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. ACM, 2006, pp. 112–121.

[30] D. Faust and C. Verhoef, "Software Product Line Migration and Deployment," *Journal of Software Practice and Experiences*, vol. 30, no. 10, pp. 933–955, 2003.

[31] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "Refactoring a Legacy Component for Reuse in a Software Product Line: a Case Study: Practice Articles," *Journal of Software Maintenance and Evolution*, vol. 18, no. 2, pp. 109–132, 2006.

[32] R. Koschke, P. Frenzel, A. P. Breu, and K. Angstmann, "Extending the Reflection Method for Consolidating Software Variants into Product Lines," *Software Quality Control*, vol. 17, no. 4, pp. 331–366, 2009.

[33] M. Esperanza Manso, J. A. Cruz-Lemus, M. Genero, and M. Piattini, "Empirical Validation of Measures for UML Class Diagrams: A Meta-Analysis Study," *MODELS 2008 Workshops, LNCS 5421*, pp. 303–313, 2009.

[34] M. Genero, J. A. Olivas, M. Piattini, and F. P. Romero, "Assessing Object-Oriented Conceptual Models Maintainability," in *ER 2002 Workshops*, 2002, pp. 288–299.

[35] M. Esperanza Manso, J. A. Cruz-Lemus, M. Genero, and M. Piattini, "Empirical validation of measures for class diagram structural complexity through controlled experiments," in *Proceedings of 5th International ECOOP Workshop on Quantitative Approaches in object-oriented Software Engineering (QAOOSE'01)*, 2001, pp. 87–95.

[36] J. Olivas and F. P. Romero, "FPKD. Fuzzy Prototypical Knowledge Discovery. Application to Forest Fire Prediction," in *Proceedings of the SEKE 2000*. Knowledge Systems Institute, 2000, pp. 47–54.

[37] J. Olivas, "Contribution to the Experimental Study of the Prediction based on Fuzzy Deformable Categories," Ph.D. dissertation, University of Castilla-La Mancha, Spain, 2000.

[38] M. Genero, E. Manso, A. Visaggio, G. Canfora, and M. Piattini, "Building Measure-Based Prediction Models for UML Class Diagram Maintainability," *Empirical Software Engineering*, vol. 12, no. 5, pp. 517–549, 2007.

[39] D. G. Kleinbaum, L. L. Kupper, and K. E. Muller, *Applied Regression Analysis and Other Multivariate Methods*, 2nd ed. Duxbury Press, 1987.

[40] J. A. Cruz-Lemus, M. Genero, and M. Piattini, "Using Controlled Experiments for Validating UML Statechart Diagrams Measures," *Software Process and Product Measurement Workshops, LNCS 4895*, pp. 129–138, 2008.

[41] J. A. Cruz-Lemus, A. Maes, M. Genero, G. Poels, and M. Piattini, "The Impact of Structural Complexity on the Understandability of UML Statechart Diagrams," *Information Science*, vol. 180, no. 11, pp. 2209–2220, 2010.

[42] R. F. Lorch and J. L. Myers, "Regression Analysis of Repeated Measures Data in Cognitive Research," *Journal of Experimental Psychology. Learning, Memory, and Cognition*, vol. 16, no. 1, pp. 149–57, 1990.