Extracting Executable Architecture From Legacy Code Using Static Reverse Engineering

Rehman Arshad,Kung-Kiu-Lau
School of Computer Science, University of Manchester
Kilburn Building, Oxford Road, Manchester, United Kingdom
e-mail: rehman.arshad, kung-kiu.lau @manchester.ac.uk

***Abstract*—Static reverse engineering techniques are based on structural information of the code. They work by building a model of abstraction that considers control structures in the code in order to extract some high-level notation. So far, most of these techniques produce abstraction models or feature locations but not the executable architecture that can transform the legacy code into modern paradigm of programming. Few approaches that extract architectural notation either require the code to be in component based orientation or lack automation. This paper presents an ongoing research that can extract executable architecture as X-MAN (component model) components from legacy code. An executable architecture contains structural and behavioural aspects of the system in an analysed manner. The extracted components can be integrated with other systems due to re-usability of the X-MAN component model. This approach neither requires the source code to be in component based orientation nor it lacks automation.**

***Keywords*—Reverse Engineering; Static Analysis; Component Based Development; Abstract Syntax Tree.**

I. Introduction

Reverse Engineering techniques are classified into *Static*, *Textual*, *Dynamic* and *Hybrid* [1]. Static techniques are based on structural information of the code. They work by building a model of states of the program and then determine all possible routes of the program at each step. Such model is called static abstraction model and it requires a fair consideration between preciseness and granularity [2]. As these techniques consider all control flows, they provide the maximum recall; this recall comes at the price of false positive results.

Reverse engineering is mostly used to extract high level abstraction models or semantics from the source code [1]. Such extraction is useful for documentation, variability management, etc., but it cannot provide an executable architecture after extraction. ”An executable architecture is a dynamic simulation of an architecture model. It captures both structural and behavioural aspects of the architecture in a form that can be visualised and analysed in a time dependent manner” [3]. A reverse engineering approach that can provide executable architecture can transform the source code into a specific notation that can be used in further implementation. In order to get an executable architecture from the source code, a technique has to consider every line of code by following the abstraction model of analysis. Textual techniques are mostly used for bug localisation or finding feature locations in the source code [4] and dynamic techniques can only produce results based on the execution trace [5]. Extracting an architecture is different form extracting high level abstraction models because an architecture has to show that every functionality exists in the original source code. Therefore, due to some important characteristics like maximum recall and minimum loss of information, static reverse engineering is the best analysis technique to consider for extracting an executable architecture from the legacy systems [2].

This paper presents an ongoing research on the extraction of executable architecture from legacy systems. The proposed technique is called Reverse Engineering X-MAN *(RX-MAN)*. RX-MAN uses static reverse engineering to extract X-MAN components [6] from the legacy code.

The remainder of this paper is organised as follows: Section II includes related work in the domain of static reverse engineering. Section III includes the basics of X-MAN component model. Section IV explains the proposed research methodology. Section V shows a simple evaluation and Section VI includes conclusion and future work.

II. Related Work

Static reverse engineering techniques can be classified by several parameters and [1] [7] [8] are some of the detailed surveys in the domain of static reverse engineering. Most of the static approaches are used for finding feature locations in the legacy systems. Some of the most well-known techniques are RecoVar [9], FLPV [10], Dependency Graph [11], Concern Graph [12], Automatic Generation [13], Language Independent Approach [14], Concern Identification [15] and Semi-Automatic Approach [16]. Out of the above-mentioned techniques, RecoVar [9] produces variability model from the source code. FLPV [10] generates code as set of optional and mandatory. Dependency Graph [11], Concern Graph [12] and Concern Identification [15] produce high level abstraction of code as graphs. Language Independent Approach [14] produces feature model from the source code. Automatic Generation [13] and Semi-Automatic Approach [16] generate a tool based view that helps in understanding the source code. All these techniques produce results in the form of high level abstraction that can help in understanding the legacy systems. Such outputs can help in analysing the system but cannot reuse the legacy code to transform it into executable architecture. Such architecture can be reused to build modern systems or can be extended to existing systems, e.g., X-MAN components can be re-composed to form family of systems and same components can be used across many systems due to modularity and separation of concerns in X-MAN component model. Such architectural output can also help against system erosion with time [17].

There are few approaches with aim to extract architecture from the source code. One of them is JAVACompExt [17] by Anquetil et al. It is a heuristic based approach that extracts Architecture Description Language (ADL) components along with the communication and services among them. This approach however requires the source code to be written with ”componentization” in mind. Componentization is the process of atomizing resources into separate reusable packages that can be easily recombined [18]. Another approach by Antoun et al. [19] re-engineers the JAVA code into Arch JAVA [20], though the process lacks automation. The approach by Chouambe et al. [21] produces composite components but the source system has to be implemented in component based notation. The presented approach in this paper is different from the above approaches because:

- It is automated.
- It does not require the source code to be in component notation.
- Unlike those approaches that are focused on architecture retrieval, our approach aims for component creation by source code transformation.

III. X-MAN Component Model

A component is defined by its unit of composition and composition mechanism, [22] e.g., in ADL, composition takes place via ports and unit of composition is an architectural unit defined as a class with provided and required services. X-MAN is different from other well-known component models because it separates control and computation unlike ADL based component models in which control and data cannot be separated and transmitted via ports together, e.g., Koala [23].

X-MAN components are defined by computation units (unit of composition) and connectors (composition mechanism). There are two types of components in X-MAN: atomic and composite. Atomic components have a computation unit and an invocation connector (composition mechanism) that acts as an interface of that component. Composite components can consist of set of atomic or composite components that are connected by composition connectors (composition mechanism). Composition connector of a composite component can be: (i) Sequencer, or (ii) Selector. A sequencer provides sequencing of atomic/composite components in a composite component and a selector provides conditional branching. All X-MAN components preserve encapsulation. Atomic components do this by encapsulate computation unit and composite do so by encapsulate computation units and composition connectors. It means composite components also preserve encapsulation of their nested components, which provides a hierarchy of encapsulated components. That is why X-MAN component model is hierarchical in nature. Further details of X-MAN component model have been discussed in [22]. Basic semantics of X-MAN component model are presented in Figure 1. Each component can have a set of services. A service is exposed functionality of a component that is used to send and receive data elements, needed for execution, e.g., A *Bank* component can have *Deposit*, *Withdraw* and *CheckBalance* services with *BalanceInformation* and *AccountNumber* as send/receive data elements.

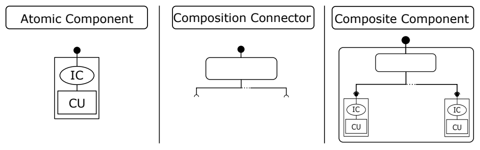

Fig. 1. X-MAN Component Model.

IV. RX-MAN: A static reverse engineering approach

Figure 2 shows the methodology of RX-MAN. Java has been selected as the language of source code to be analysed. The source code to be reverse engineered can be: (i) A single application system (ii) Just a set of classes, e.g., any library or SDK that cannot be executed on its own.

In case of single application system, output will be a one big X-MAN system that will show the whole functionality of the source code with the benefits of modularity and hierarchy of X-MAN component model. Control structures in the source code will be transformed as composition connectors. In case of the source code which is just a SDK or a set of classes, the output will be X-MAN atomic components. These components can be deposited to X-MAN repository and can be recomposed for further implementation. The brief overview of the whole process is as follows:

1) AST Tree Generation: Abstract Syntax Tree (AST) is a powerful parser in JAVA. In this step, the source code is transformed into AST nodes. Each node is mapped to its respective sub nodes, e.g., each package node is mapped to its class nodes, each class node is mapped to its method nodes and each method node is mapped to its parameters, return node, function type node etc. Detailed algorithm is not given due to its voluminous details.

2) Parsing the Nodes: AST allows the code re-writing in order to implement small changes in the code. However, AST re-writing is not powerful and convenient enough to transform the system into some other complex notation. Therefore, an intermediate data structure has been used to extract information from the nodes to preserve it in a meaningful notation. In this step, invocations of all methods are indexed and mapped against each other.

Algorithm 1 METHOD ALLOCATION

```
Require: PackageClassList, MethodClassList, utilityComponentList
  while $i < MethodClassList$ do
    if $Mi.Visited \leftarrow False$ then
      Get Invocations of Mi
      if $Mi.invocations$ is NULL then
        Mi.getPackageName
        if $Mi.PackageName$ already exists then
          AddU(PackageName,Mi)
          $Mi.visited \leftarrow TRUE$
        else
          Create U(PackageName)
          AddU(U,Mi)
          $Mi.visited \leftarrow TRUE$
        end if
      else
        ExtractEachInvocation(Mi)
      end if
    end if
  end while
```

Fig. 3. Component-Method Allocation.

3) Static Abstract Model of Abstraction: This step shows the first cycle of reverse engineering. This step maps the rules to create X-MAN atomic components and then assign methods to components based on the rules of allocation. One important parameter that has to be defined is size of the component. Size of the components should be realistic. If an approach extracts 10 components from the source code with only 7 classes then it does not justify the use of components. Similarly, one big component that represents 50 classes is not ideal either. There are two ways in which the component size can be defined in our tool: (i) Package Based Restriction (ii) Number of methods in each computation unit. Depending on the source code, a reasonable restriction can be applied to limit the number of methods in each computation unit. The package based restriction is compatible with JAVA because packages are usually created and designed to differentiate specific set of tasks. Package based restriction does not mean that a method M1 in class C1 of package P1 always belongs to the component of P1. It means that the maximum number of extracted components cannot exceed the total number of packages in the code, where the minimum number of components that can be extracted is 1. Method M1 can belong to any component depending on the rules of algorithm of allocation.

Algorithm 1 shows the start of allocation. *MethodClassList* has all the methods we have extracted from AST nodes and stored in our data structures. Similarly, *PackageClassList* has all the classes against their packages. Additional information like method parameters, return types and method invocation list can also be retrieved by using a *HashMap* against each index of these lists. Invocation list of each method will be matched and the methods with zero invocations will be considered as utility components. Such methods are not invoking any other method, it means they are mostly conducting simple tasks for other methods but do not require anything from any other method in the source code. Such methods will be placed

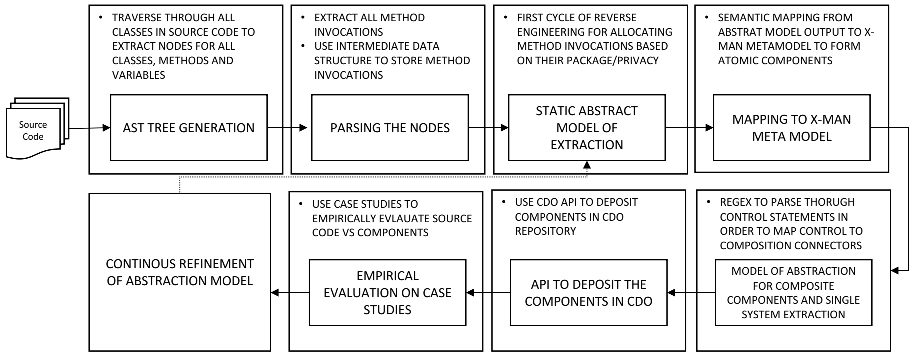
Source Code
TRAVERSE THROUGH ALL CLASSES IN SOURCE CODE TO EXTRACT NODES FOR ALL CLASSES, METHODS AND VARIABLES
AST TREE GENERATION
EXTRACT ALL METHOD INVOCATIONS
USE INTERMEDIATE DATA STRUCTURE TO STORE METHOD INVOCATIONS
PARSING THE NODES
FIRST CYCLE OF REVERSE ENGINEERING FOR ALLOCATING METHOD INVOCATIONS BASED ON THEIR PACKAGE/PRIVACY
STATIC ABSTRACT MODEL OF EXTRACTION
SEMANTIC MAPPING FROM ABSTRAT MODEL OUTPUT TO X-MAN METAMODEL TO FORM ATOMIC COMPONENTS
MAPPING TO X-MAN META MODEL
REGEX TO PARSE THORUGH CONTROL STATEMENTS IN ORDER TO MAP CONTROL TO COMPOSITION CONNECTORS
MODEL OF ABSTRACTION FOR COMPOSITE COMPONENTS AND SINGLE SYSTEM EXTRACTION
USE CDO API TO DEPOSIT COMPONENTS IN CDO REPOSITORY
API TO DEPOSIT THE COMPONENTS IN CDO
USE CASE STUDIES TO EMPIRICALLY EVLAUATE SOURCE CODE VS COMPONENTS
EMPIRICAL EVALUATION ON CASE STUDIES
CONTINOUS REFINEMENT OF ABSTRACTION MODEL

Fig. 2. RX-MAN Methodology.

Algorithm 2 Extract Each Invocation

```
Require: Mi,index
  while $Mi.invocationList! = Empty$ do
    if $Mi.PackageName == Mi.invocationList(index).packageName$ then
      if $C.PackageName \leftarrow FALSE$ then
        CreateComponent(PackageName)
      end if
      CheckDuplication(Mi,Mi.invocationList(index),PackageName)
      if $Duplication == FALSE$ then
        ADD (Mi,Mi.invocationList(index),PackageName)
      end if
      ExtractEachInvocation(Mi.invocationList(index),index)
    else
      if $Mi.invocationList(index).package \leftarrow FALSE$ then
        CreateComponent(PackageName)
      end if
      CheckDuplication(Mi,Mi.invocationList(index),PackageName)
      if $Duplication == FALSE$ then
        CheckPrivateMemberAccess(Mi.invocationList(index))
        if $CheckPrivateMemberAccess \leftarrow FASLE$ then
          ADD (Mi,Mi.invocationList(index),PackageName)
        else
          SetAllocation(Mi.invocationList(index).PackageName)
        end if
        ExtractEachInvocation(Mi.invocationList(index),index)
      end if
    end if
  end while
```

Fig. 4. Methods Invocations Extraction.

in utility components. Each X-MAN component will have its own utility component in which all such methods will be placed. This approach will help in reducing the coupling in the original source code. Method *createU* creates a utility component if a utility function belongs to a X-MAN component and method *AddU* adds a utility function in a utility component if it already exists. All other methods will be considered to place in X-MAN atomic components and their invocations will be extracted for further allocations.

In algorithm 2, for each method, its invocation list is extracted and compared with it. If the method Mi and its invoked method belongs to same package, then a component with that package name is created and both methods will be placed in computation unit along with their imports and class variables they use. If both belong to different packages, then a privacy check will be conducted by function *CheckPrivateMemberAccess*. If the method being invoked accesses the private variables or calls private functions in that package in its invocation list, then that method cannot be placed with method Mi. In that case, the invoked method will be placed in a newly created component (if does not already exist) along with the private methods it is accessing (*SetAllocation()* in Algorithm 2). Same process will be applied to all the invoked methods in the invocation list of Mi. Several factors have to be considered before placing a method at appropriate location, e.g., its access to global variables, usage of its local variables in other private methods etc. Function *Duplication* checks whether the method being invoked is already part of the component. At the end of this cycle, each method will be placed in appropriate component in a notation which will be mapped to X-MAN meta model. A user can select any combination of the public methods in a computation unit as a service for that component. For *Number of methods in each computation unit* approach, rules will be applied based on the number of methods in each computation unit and not on the package based allocation.

4) Mapping to X-MAN MetaModel: Extracted results are mapped to X-MAN meta-model. This step also involves the X-MAN validation to make sure only valid X-MAN components can be deposited. Validation involves the semantic checks against X-MAN meta-model in order to check that there is no violation against the semantics of X-MAN component model. X-MAN meta-model is presented here [24].

5) Single System Extraction: For single application systems, second cycle of reverse engineering is needed in order to extract composite components and composition mechanism among them. So far, we have considered *if-else*, *While*, *For* and *Switch* statements as candidates of composition connectors. They can appear in any combination hence each possible scenario should be mapped to X-MAN semantics. Few examples of such mapping are shown in Figure 5. This part of research is theoretically completed but still under development in our tool.

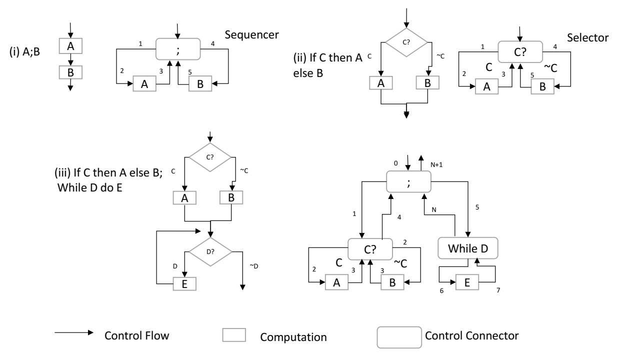

Fig. 5. Control Structures VS X-MAN Equivalent.

6) CDO Repository: CDO is a framework with development time model repository. It is implemented along with EMF (Eclipse Modelling Framework) in implementation of X-MAN tool. Every extracted component can be deposited, retrieved and recomposed according to needs.

TABLE I. RX-MAN INITIAL RESULTS

Application	**Classes**	**X-MAN Components**	**Component To Class Size %**
JabRef	35	8	4.3%
TeamMates	51	4	12.7%
EverNote	27	4	6.75%

V. Evaluation

So far, we have applied our approach to extract atomic X-MAN components on the variety of JAVA based projects. These results are output of the first cycle of reverse engineering, whereas second cycle implementation is in development. Three most notable application we have used are Jabref [25], Evernote-sdk and Team-mates. Jabref is a well-known database management tool and widely used by researchers along with latex. TeamMates is a free online tool for managing peer evaluations and Evernote is a famous cross platform app. All are open source JAVA based projects.

Table I shows the results of R-XMAN. The result is quite diverse and depends on the nature of the code. In case of Jabref, we got 8 components out of 35 classes (it means each component has average size equal to 4.3 classes of the original source code) and in case of Team-mates, we got 4 components out of 51 classes. We have only used that part of the code which is related to model and middle layer of the applications. In case of Evernote-sdk, we got 4 components out of 27 classes. Figure 6 shows R-XMAN tool with extracted components of JabRef. The main panel shows two

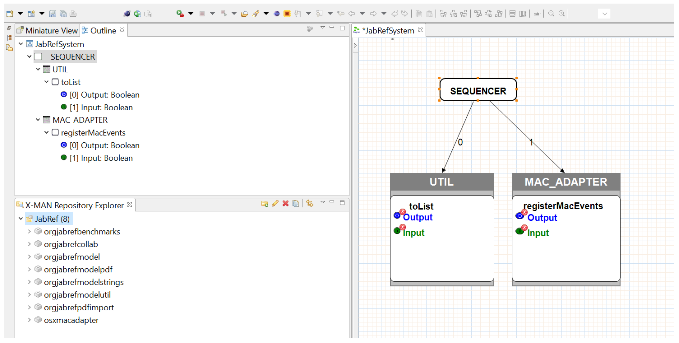

Fig. 6. RX-MAN Tool.

of the extracted components composed by sequencer. Sequencer will execute Route 0 before the route 1 hence *UTIL* component will be executed before *ADAPTER* component. *registerMacEvents* and *toList* are services of these components.

An important point to consider here is that the X-MAN is a model for computation, not a model for resource allocation. Of course, we can study resource and memory problems for X-MAN systems but major thing to consider here is that the components will be stored only once, but reused many times. Therefore, we need less memory overall.

VI. Conclusion and Future Work

RX-MAN is a static reverse engineering approach that can extract executable architecture from source code as X-MAN components. So far, we have implemented the first cycle of reverse engineering. Comprehensive evaluation of the methodology demands the completion of second cycle of reverse engineering in order to compare the extracted system with original source code. The approach has been applied on several small examples, but significant case studies are needed for further evaluation. We have picked *Qualitus Corpus* [26] for evaluation. *Qualitus Corpus* is a well-known collection of JAVA systems for empirical studies and three systems will be selected for evaluation. Further future work includes the integration of reverse engineering with software product lines in order to achieve product line architecture from legacy systems.

Overall, there are the following benefits for selecting X-MAN over other component models as an output notation of reverse engineering:

- Separation of control (composition connectors) and computation (computation unit).
- Ability to compose in both design and deployment phase of component life cycle. One can deposit, retrieve, re-compose and tailor X-MAN components according to needs where it is not possible with ADL based component models.
- No required services like ADL based component models due to exogenous composition.

References

[1] K.-K. Lau and R. Arshad, *A Concise Classification of Reverse Engineering Approaches for Software Product Lines*. 4 2016.

[2] M. D. Ernst, “Static and dynamic analysis: Synergy and duality,” in *WODA 2003: ICSE Workshop on Dynamic Analysis*, pp. 24–27, Citeseer, 2003.

[3] P. Helle and P. Levier, “From integrated architecture to integrated executable architecture,” in *Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), 2010 19th IEEE International Workshop on*, pp. 148–153, IEEE, 2010.

[4] D. Poshyvanyk and A. Marcus, “Combining formal concept analysis with information retrieval for concept location in source code,” in *Program Comprehension, 2007. ICPC’07. 15th IEEE International Conference on*, pp. 37–48, IEEE, 2007.

[5] A. D. Eisenberg and K. De Volder, “Dynamic feature traces: Finding features in unfamiliar code,” in *Software Maintenance, 2005. ICSM’05. Proceedings of the 21st IEEE International Conference on*, pp. 337–346, IEEE, 2005.

[6] K.-K. Lau, L. Safie, P. Stepan, and C. Tran, “A component model that is both control-driven and data-driven,” in *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*, pp. 41–50, ACM, 2011.

[7] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, “Feature location in source code: a taxonomy and survey,” *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[8] M. L. Nelson, “A survey of reverse engineering and program comprehension,” *arXiv preprint cs/0503068*, 2005.

[9] B. Zhang and M. Becker, “Recovar: A solution framework towards reverse engineering variability,” in *Product Line Approaches in Software Engineering (PLEASE), 2013 4th International Workshop on*, pp. 45–48, IEEE, 2013.

[10] Y. Xue, Z. Xing, and S. Jarzabek, “Feature location in a collection of product variants,” in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pp. 145–154, IEEE, 2012.

[11] K. Chen and V. Rajlich, “Case study of feature location using dependence graph.,” in *IWPC*, pp. 241–247, Citeseer, 2000.

[12] M. P. Robillard and G. C. Murphy, “Concern graphs: finding and describing concerns using structural program dependencies,” in *Proceedings of the 24th international conference on Software engineering*, pp. 406–416, ACM, 2002.

[13] M. P. Robillard, “Automatic generation of suggestions for program investigation,” in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 11–20, ACM, 2005.

[14] T. Ziadi, C. Henard, M. Papadakis, M. Ziane, and Y. Le Traon, “Towards a language-independent approach for reverse-engineering of software product lines,” in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pp. 1064–1071, ACM, 2014.

[15] M. Trifu, “Improving the dataflow-based concern identification approach,” in *Software Maintenance and Reengineering, 2009. CSMR’09. 13th European Conference on*, pp. 109–118, IEEE, 2009.

[16] M. T. Valente, V. Borges, and L. Passos, “A semi-automatic approach for extracting software product lines,” *Software Engineering, IEEE Transactions on*, vol. 38, no. 4, pp. 737–754, 2012.

[17] N. Anquetil, J.-C. Royer, P. Andre, G. Ardourel, P. Hnetynka, T. Poch, D. Petrascu, and V. Petrascu, "Javacompext: Extracting architectural elements from java source code," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pp. 317–318, IEEE, 2009.

[18] "What do we mean by componentization (for knowledge)? – open knowledge international blog," April 2007. (Accessed on 07/31/2017).

[19] M. Abi-Antoun, J. Aldrich, and W. Coelho, "A case study in re-engineering to enforce architectural control flow and data sharing," *Journal of Systems and Software*, vol. 80, no. 2, pp. 240–264, 2007.

[20] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *Proceedings of the 24th international conference on Software engineering*, pp. 187–197, ACM, 2002.

[21] L. Chouambe, B. Klatt, and K. Krogmann, "Reverse engineering software-models of component-based systems," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pp. 93–102, IEEE, 2008.

[22] K.-K. Lau, L. Safie, P. ˇ Stˇ epán, and C. Tran, "A component model that is both control-driven and data-driven," in *Proc. 14th Int. ACM SIGSOFT Symp. on Component-based Software Engineering, LNCS 6092*, pp. 41–50, ACM, 2011.

[23] T. Asikainen, T. Soininen, and T. Männistö, "A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families," in *Software Product-Family Engineering*, pp. 225–249, Springer, 2004.

[24] K.-K. Lau and C. M. Tran, "X-man: An mde tool for component-based system development," in *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pp. 158–165, IEEE, 2012.

[25] M. Alver, N. Batada, M. Baylac, K. Brix, G. Gardey, C. D'Haese, R. Nagel, C. Oezbeck, E. Reitmayr, A. Rudert, *et al.*, "Jabref reference manager," 2003.

[26] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pp. 336–345, IEEE, 2010.