Towards a Code Synchronization Mechanism Analysis and Design

Ozan Aksoy
Department of Computer Science & Department of Software Engineering
Uppsala University & Tongji University
Uppsala, Sweden & Shanghai, China
Ozan.Aksoy.0767@student.uu.se

***Abstract* — This paper presents the first steps of an endeavor for defining and applying equalization among code artifacts. This equalization process is defined as code synchronization. First, meaning of the code synchronization has been explained, and also the constraints on the synchronization as a process have been analyzed. On the following, four major necessary abilities for realizing a code synchronization mechanism have been identified. Two of the four abilities that are identified as necessary for creation and application of the previously defined synchronization process have been discussed. Lastly conclusions from these discussions have been underlined.**

***Index Terms* —** Code Synchronization, Software Evolution, Automated Software Engineering, Change Extraction

I. INTRODUCTION

One of the realities during software development is the change in existing code structures. From the beginning of the development process, there is constant attempt to change the code bodies. Therefore, software is defined as an evolving structure [1]. The complexity of the relations among code artifacts are affected in different levels. These effects are dependent to the nature of relation that defines these structures from these changes. It is claimed that these effects can be traced and analyzed [2].

At the first phases of software development, most of the effort is spent on the documentation and the design of the project [3]. On later phases of development, the design began to lack the consistency with the changes of the implementation level. As it can be seen from the literature review of [4], these consistency issues can manifest itself throughout the development process in many forms. Identification and resolution of these changes are resolved by using mechanisms that provide *traceability on code artifacts*, and *analysis of impact from the changes* on related code and model. This impact of change is studied in software structures that are designed in object oriented architecture in various prior studies [4].

Beyond keeping consistency and maintenance of evolving software, if one's aim is to create an equivalent of a code body in one domain to a corresponding another domain, then the requirements are in need of a research. This attempt of applying an equalizing process can be defined as "synchronizing" the code bodies. Thus, we can define "code synchronization" process as bringing a unified form to a code body. This requires understanding the structure of the code. Moreover, it can be claimed that such a structure is also the equivalent of the model of the code [5].

As explained in [6], Modeling is a concept which has many forms in daily practices. If one were to think the entities around us according to our perception, it can be said that their defining features could be described differently. We can call these descriptions as the idealized form of an entity, where 'idea' is also an entity by itself. Moreover, if these entities obtained behavior or properties, they can be defined as owning a system or a part of the system which includes a sub-system. Thus, systems that cover other sub-systems are acting like abstract forms with sub-behavioral steps. These steps form a workflow which realizes the abstract behavior.

As in Meta Model concept, it is significant to point that for modeling it is the architectural background that carries such an abstract definition of a system, where this abstraction is the 'idea' of designed entity, such as a domain specific language (DSL) [7]. Using this idea, we can create logical definitions of the perceived systems, such as new domain languages that are modeled on a Meta Model level. These Meta Models are then applied to a framework as the grammar of the created language. Apparently, this is true for all systems that apply object oriented architecture. Objects are defined with methods and attributes that they own, as it is defined in object oriented modeling standards. Their relations are defined in layers of abstraction, where every object is a class definition and a class is the scheme which holds the blueprint of the object, and this schemes can be traced among correspondent code bodies [8]. Therefore, it can be stated that a model of the code body and the architecture that it depends implicitly forces a definable, comparable form to code bodies that are created in object oriented design. Bringing equivalent form and behavior to code bodies requires extracting such limitations, expected situations and exception situations.

It can be said that "synchronization" or "code synchronization" is born out of a process that occurs between evolving structural states of corresponding code bodies. Such an attempt tries to bring an equalized state among compared code bodies. Moreover, these code bodies can have different design structures. They can be acting as a model scheme or a source code body in their relation as correspondents.

Such a synchronization process has to be configured in order to behave the way that is desired. This necessity is born because two states of compared code bodies can achieve a synchronized relation in various ways. Such ways includes e.g. loss of information to bring an equalized form, partially equalizing code bodies to each other, a complete equalization by generation of all corresponding elements. Therefore, behavior of the synchronization process is a key factor that determines the outcome.

For implementation of the synchronization process in the target development environment, a synchronization mechanism has to be tailored. Thus, it is a necessity to identify the research topics and related techniques for realizing such a mechanism.

II. Defining Necessary Abilities for realizing Synchronization Mechanism

Defining synchronization as a process requires identifying the abilities that is necessary to catch the changes on the code bodies. Therefore, in order to find the necessary abilities for synchronization attempts, five major issues have been analyzed. These five issues are:

- How to relate the correct code parts to each other?
- How to generate the equivalent code in correlated code domains?
- How to understand the changes on the code domain?
- How to understand which part to change in order to reflect the changes on corresponding domain?
- How to conduct the synchronization implementation decision?

Each issue has been studied and linked to a solution. First issue is related to traceability of code artifacts. Second issue is related to generation ability, and it depends on the functional relation and semantic relation of the code bodies. Third and fourth issue is a matter of understanding the impact of changes from one or more code artifacts to correlated code artifacts. Finally, the fifth issue must be solved via introducing a behavior analysis method which is used in predicting the outcome of a synchronization implementation, thus setting an expected behavior.

Though a behavioral analysis, synchronization attempt is calculated before its implementation, which gives a holistic outcome of the attempt. Thus, these calculations act as the decision maker or – the brain – of synchronization mechanism. This decision making process is necessary for avoiding undesired outcomes and realization of desired synchronization behavior, which leads to the correct implementation.

A. Required Abilities for Realizing Synchronization Mechanism

From the five major issues of synchronization, a synchronization mechanism can be summarized in four abilities as listed below.

- *Tracing Ability:* It is used in relation of code parts and storing or extracting this relation information
- *Change Notification Ability:* It understands and notifies about the changes on related code bodies.
- *Generation Ability:* It checks whether if equivalent code can be generated, also implements the decided behavior.
- *Decision Making Ability:* It analyzes the whole process and decides the outcome and implementation decision.

B. Identifying Required Sub-Mechanisms for Achieving Necessary Abilities

As a result of such a classification, a field research has been conducted, and existing solutions for the identified necessary abilities have been studied.

For *tracing ability*, traceability mechanisms have been overviewed and studies that are found to be in similar grounds and studies that have suggested interested solutions to synchronization issues have been analyzed.

For *change notification ability*, studies that are interested in impact of change and analysis have been aggregated and interesting results have been noted.

For *generation ability* and for *decision making ability*, further research will be conducted. However, they are not included in this paper.

III. Discussion on Traceability for Synchronization

There are hardships of creating and maintaining traceability. It is mostly because the need for knowledge gathering and sharing in a setting where dependent bodies and mechanisms work together leads to a chaos on the mechanic demands [9]. Depending on the complexity of the technique, one can enhance the abilities of the traceability mechanisms.

One repeating issue in traceability solutions is that, traceability mechanisms end up being forced to solve inconsistency that occurs on tracing code itself. This occurs because of differences in compared structures, and software evolution phase of tracing code itself. Such findings, and also solution suggestions for these issues that are aimed for both collection and maintenance of traceability information, have been reported in [10].

Moreover, a tracing mechanism can be realized with internal information storage design or external information storage design. In the research of [11], it is suggested that there are advantages to use external traceability mechanisms over internal tracing mechanisms. In their study, it was stated that internal storage of traceability links means to store the tracing links on target domains. This approach causes:

- If the link is directed and stored in the source model only, it is not visible from the target model.
- If traceability information is stored in the both models, then this information must be maintained for consistency.
- Adding internal links to code may cause too much inclusion of secondary importance code parts, thus creating readability problems.
- Additional internal traceability links can cause breaking the natural form of code domain.

On the other hand, external traceability systems are claimed to have advantages as:

- Code pollution, which makes readability harder because internal traceability links, is avoided.
- It is easier to extract data and maintain the links on

automated mechanisms.

- In order to create an external traceability mechanism, it is required to have referenced models with unique identifiers.

There is a particular guiding source that was found during this research. In his study in surveying various traceability techniques [2], Schwarz addressed the six different problem domains that traceability mechanism aimed to bring solutions. These are:

1) Definition: defining the target domain to establish traceability

2) Identification: discovering relations of artifacts in target domain

3) Recording: how to keep track of the trace relations

4) Retrieval: how to access to related artifacts using the chosen record technique

5) Utilization: additional editing and management utilities for tracing mechanisms.

6) Maintenance of traceability relationships: how to re-establish lost trace records.

By these classifications, we can structure the necessary mechanical requirements for synchronization attempt.

B. Studies aimed to create and maintain traceability via analysis in context of code

In the work of [12], researchers had worked on synchronizing the use case models (UCMs) with implementation models, thus creating a model-to-model traceability mechanism. Researchers aimed to keep abstract system model of UCMs and implementation model which is defined by unified modeling language (UML) in synch. Since implementation model is matched by the Java Source Code, the mechanism is also able to keep the abstract model updated through java source code. Moreover, implementation model can be inferred from the Java Code and then matched to the UCMs.

Another approach comes from the work of [13]. In this work, objects in object oriented design paradigm have been defined as *evolving*. In order to catch the difference that occurs as the development continues, a similarity catcher has been proposed. This mechanism cleans the noise on the code artifacts and processes the source code to derive a conclusion about the similarity of the code artifacts. A matcher mechanism with *maximum similarity* approach is used in iterations to catch the matching pairs.

An alternative perspective to traceability creation can be found in [9]. In this work, code context is revealed via questioning the developers and inferring the desired relations. This approach is unique in its narrative that traceability creation is left to developer's answers in creating a context relation graph. Developers are perceived as the main attention for the framework. However, this study is aimed to trace use-cases to code artifacts. Therefore, this approach can be seen as a specific domain solution.

A study that appeared on the scene as one of the early approaches to traceability problem came from [14]. In his approach, Egzyed tests the user scenarios versus implementation for the project. Via a comparison mechanism that was suggested to fit the requirements, it was aimed to create links from model to source code.

C. Studies aimed to create traceability links via lightweight links

In the work of [15], researchers have proposed a plug-in for Eclipse Platform. In the Plug-in, they aimed to present similarity level between identified source code identifiers. In this case, instead of traceability links, mechanism tries to match the similarity from the key identifiers on the code.

In the work of [5], researchers have developed a DSL language in order to trace the related artifacts during the evolution of the software, such as documentation and corresponding source code. In order to achieve this, they used a mapping technique which they call "feature maps". These maps are composed via tiny code statements in annotation. This links are used in a revision control system. This revision control system is similar to a version control system; it uses the comparison mechanism that is also used in UNIX file systems.

In [16], use of XML to create the traceability framework has been analyzed and developed. In brief, textual files have been transformed to XML language representations. Though these representations, any model can be traced to the each other. XML language representations act as a common and comparable form. This technique resembles XML schema use in Eclipse Modeling Framework (EMF) [17]. In EMF, object specifications are converted to XMI, and later linked to corresponding source code. Xtext also uses XMI in order to link the rules and constraints that cannot be defined just by using the EMF models.

Another XML based approach to traceability was [3], which uses XML based lightweight links to create a feature mapping which is similar to approach in [5]. They also emphasize the need for internal structural requirements for traceability via links. According to this study, the most suitable condition for a traceability link to work is a fine-grained design with syntactic differencing. This is an expected outcome, since a major problem for lightweight links are losing their mapped relations during software evolution [18]. Controlling the consistency of such lightweight link approaches causes additional maintenance problems during the evolution of the software projects. These problems are addressed in various researches, such as [2], [13], [16].

IV. Discussion on Change in Code Artifacts for Synchronization

Tracing mechanisms alone cannot be sufficient to ensure the realization of the synchronization ability. This is because of the need to understand the impact of change that will occur on the artifacts. This impact greatly varies among the artifacts. Many researchers focus on the impact of change in various forms. Thus, for synchronization mechanisms, another step for realization of the synchronization is to understand the impact of change and making a decision for required change on other related parties which is affected by the change.

From the perception of this research, impact of change has been constructed in two major concepts. These are the comparison of artifacts, and the initialization of analysis.

All kind of change analysis types need a comparative relation to some other form of information that is related to either some form of itself or another information block that it can relate. This step, which is simply referred as comparison of artifacts, is similar to implementations in version control mechanisms or repository control mechanisms, such as Control Versioning System (CVS), Apache Maven Project, or GitHub Service. However, in a mechanism such as EMF Compare, one can see implementation for general comparison systems that can relate to any form of supported models; in this case, ECore models. However, the impact of change analysis capabilities of such repository analysis alone is not sufficient for artifact comparison for synchronization among code artifacts from different roles and structures.

As mentioned before, second aspect for impact of change analysis is the initialization of the analysis. This is interested with the time that the comparable information forms for code artifacts are compared to each other. This is important because it directly refers to what versions the mechanism has been analyzed. Incorrect selection for time of analysis will lead to false or unusable results.

From the studies of De Lucia, Fasano and Oliveto [10], one can conclude that, for the impact of change analysis, it is important to consider:

- Compared states
- Time of Analysis
- Deriving correct decision
- Implementation of Decision

Moreover, it was stated that the analysis of impact of change and traceability are interrelated to each other [10].

For this research, one of the most influential studies was [19]. In this study, Omori and Maruyama presented their findings on a plug-in for Eclipse IDE. This plug-in is called "OperationRecorder". Omori and Maruyama's aim was to develop a mechanism that stores the individual changes that occur while software development. They present three key points in their research:

1) Instead of comparing two code bodies to each other, such as versions in repositories, they create ASTs from successfully compiled code files. These ASTs are then related to an edit history of an individual item. Thus, they create the possibility to reverse individual items to a form where they can be successfully compiled again.

2) They try to implement a mathematical mechanism to decide whether if the AST items are related to each other. They use a similarly catcher mechanism which is based on the offsets of the entities by scoring them from an initial start where editing began until the current AST item. This creates the possibility to find relations in complex modifications in the code body. However they have concerns about the success rates in high complexities. This seems especially true for text that was both modified and replaced.

3) They warn the users for possible performance hindering on IDE's computation speed.

Thus, study of Omori & Maruyama displays relations between the code elements in an individual code entity. There are also studies that implemented additional abilities on findings of [19] in [20]. This enhancement is achieved by adding an annotation mechanism that can be used by developers to insert editing information for clustering the editing records in correct order, thus creating a finer result in catching changes. However, in case where editing record is not available, there is also a methodology that can extract the editing information from two versions of related ASTs. This method has been constructed by [21]. This approach requires a hierarchic order, and thus fits to the nature of ASTs. A study that targets to use this method exists in [18].

On the other hand, in [22] they have focused on classes, and they have developed a mechanism which uses vector-space comparison computations. In this kind of comparison, two classes are syntactically analyzed to create vector values according to identified text that is repeated in the class. In the research of [22], we can see two main results:

1) Vector-space computation does not have the capability to map code elements (properties and operations), but rather creates heuristic comparison results via matching code entities (classes). Code elements can be added, removed, changed but they represent a smaller scale of impact to a class consistency, as described in [23]. For a code entity, impact of changes on the class identification itself has larger impact.

2) Possible evolutionary path of a code entity has been presented. There are seven possible change behaviors for a class. These are defined as: *Replacement*, *Extraction*, *Merge*, *Split*, *Merge into a new class*, *Recombine*, and *Recombine into new classes*.

Additional to comparison methodology at the level of code elements, and comparison methodology among code entities; there is a need for a methodology for comparison among entities of different modeling forms for complete coverage of synchronization attempts. An important influence to solve these cases is studied in [24]. In their work, they propose a technique to create a XML representation for comparing models and source code via a difference catcher method that is commonly used in versioning systems. Briefly, like Omori & Maruyama, they create an AST and convert it to XML versions which later used in various model to model or model to text comparisons.

V. Conclusion

Findings from studies that are related to traceability and impact of change topics have provided the first steps of an overview for designing the synchronization mechanism. Considered traceability studies indicate that the storage of the tracing link information has an impact on the workflow and the durability of the software. For synchronization mechanisms, two traceability methods have been identified to be in value. These are conducted thought two major application styles. First technique achieves traceability by using links that carry the relation information among code artifact. A second option is to infer the relations by *on the scene* analysis of the code artifacts.

Moreover, two major findings have been deduced from studies that are related to impact of change analysis. First finding is the four main points for identifying the types of changes on code artifacts. This information is useful for defining the aims to be satisfied during the construction of a mechanism that realizes the change notification ability. Secondly, definitions for impact of change provide the analysis of expected outcomes from the changes. This is important to detect the limited, expected and exception conditions.

A. Future Work

In further studies, both generation ability and the synchronization behavior are planned to be researched in detail. After such an investigation, the aim target is to find solutions to issues for all the four sub-mechanisms. Later, a complete design methodology will be constructed by using the findings from those solution attempts. On the following phases of the research, a synchronization mechanism is aimed to be designed and documented with previously defined design methodology.

ACKNOWLEDGMENT

I would like to thank to BMW China, and both of my universities, Uppsala University and Tongji University, for giving me the opportunity to start this research as a part of a master thesis.

REFERENCES

[1] Lehman, "Programs, life cycles, software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060- 1076, September 1980.

[2] Hannes Schwarz, "Towards a comprehensive traceability approach in the context of software maintenance," in *13th European Conference on Software Maintenance and Reengineering*, Kaiserslautern, 2009, pp. 339-342.

[3] Jonathan I. Maletic, Michael L. Collard, and Bonita Simoes, "An XML based approach to support the evolution of model-to-model traceability links," in *TEFSE '05 Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, Long Beach CA, USA, 2005, pp. 67-72.

[4] Hans Christian Benestad, Bente Anda, and Erik Arisholm, "Understanding software maintenance and evolution by analyzing individual changes: a literature review," *Journal of Software Maintanance and Evolution: Research and Practice*, vol. 21, no. 6, pp. 349–378, September 2009.

[5] Grady Booch et al., *Object-oriented analysis and design with applications*, Third Edition ed., Clemens Szyperski, Ed. Massachusetts, United States: Pearson Education, Inc., 2007.

[6] Colin Atkinson and Thomas Kuhne, "Model-driven development: A metamodeling foundation," *IEEE Computer Society Press*, vol. 20, no. 5, pp. 36-41, September 2003.

[7] Richard C. Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*, 1st ed., Gamma Erich, Nackman Lee, and Wiegand John, Eds. Massachusetts, United States of America: Pearson Education, Inc., 2009.

[8] Inah Omoronyia, Guttorm Sindre, Marc Roper, John Ferguson, and Murray Wood, "Use case to source code traceability: the developer navigation view point," in *17th IEEE International Requirements Engineering Conference*, Atlanta, GA, 2009, pp. 238-242.

[9] Andrea De Lucia, Fausto Fasano, and Rocco Oliveto, "Recovering traceability links in software artifact management systems using information retrieval methods," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 4, September 2007, Artical 13, 50 pages.

[10] Nicholas Drivalos, Richard F. Paige, Kiran F. Fernandes, and Dimitrios Kolovos, "Rigorous identification and encoding of trace-links in model-driven engineering," *Journal of Software and Systems Modeling*, vol. 10, no. 4, pp. 469-487, October 2011.

[11] Jorge Andres Diaz-Pace, Juan P. Carlino, Martin Blech, Alvaro Soria, and Marcelo R. Campo, "Assisting the synchronization of UCM-based architectural documentation with implementation," in *European Conference on Software Architecture WICSA/ECSA, Joint Working IEEE/IFIP Conference*, Cambridge, UK, 2009, pp. 151-160.

[12] Giuliano Antoniol, Alessandra Potrich, Paolo Tonella, and Roberto Fiutem, "Evolving object oriented design to improve code traceability," in *Proceedings of the 7th International Workshop on Program Comprehension*, Pittsburgh, 1999, pp. 151-160.

[13] Alexander Egyed, "A scenario-driven approach to traceability," in *Proceedings of the 23rd Interantional Conference on Software Engineering*, Toronto, 2001, pp. 123-132.

[14] Jim Alves-Foss, Daniel Conte de Leon, and Paul Oman, "Experiments in the use of XML to enhance traceability between object-oriented design specifications and source code," in *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)*, vol. 9, Hawaii, 2002, pp. 276-283.

[15] Steinberg Dave, Budinsky Frank, Paternosto Marcelo, and Merks Ed, *EMF: Eclipse Modeling Framework*, 2nd ed., Gamma Eric, Nackman Lee, and John Wiegand, Eds.: Addison-Wesley Professional, 2008.

[16] De Lucia Andrea, Di Penta Massimiliano, Rocco Oliveto, and Zurolo Francesco, "COCONUT: code comprehension nurturant using traceability," in *22nd IEEE International Conference on Software Maintenance*, Philadelphia, 2006, pp. 274-275.

[17] Sukanya Ratanotayanon, Susan Elliott Sim, and Derek J. Raycraft, "Cross-artifact traceability using lightweight links," in *Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, Vancouver, Canada, 2009, pp. 57-64.

[18] Michael Würch, Martin Pinzer, Harald C. Gall, and Beat Fluri, "Change distilling: tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725-743, November 2007.

[19] Takayuki Omori and Katsuhisa Maruyama, "A change-aware development environment by recording editing operations of source code," in *Proceedings of the 2008 international working conference on Mining software repositories*, New York, NY, USA, 2008, pp. 31-34.

[20] Shinpei Hayashi and Motoshi Saeki, "Recording finer-grained software evolution with IDE: an annotation-based approach," in *IWPSE-EVOL '10 Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, New York, NY, 2010, pp. 8-12.

[21] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom, "Change detection in hierarchically structured information," in *SIGMOD '96 Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1996, pp. 493-504.

[22] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo, "An automatic approach to identify class evolution discontinuities," in *IWPSE '04 Proceedings of the Principles of Software Evolution, 7th International Workshop*, Washington, DC, 2004, pp. 31-40.

[23] Maen Hammad, Collard L. Michael, and Jonathan I. Maletic, "Automatically identifying changes that impact code-to-design traceability during evolution," *Software Quality Journal*, vol. 19, no. 1, pp. 35-64, March 2011.

[24] Nikolaos Tsantalis, Natalia Negara, and Eleni Stroulia, "Webdiff: a generic differencing service for software artifacts," in *ICSM '11, Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, Washington, DC, 2011, pp. 586-589.