Managing Traceability Links With MaTraca

Angela Lozano
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
Email: alozano@soft.vub.ac.be

Carlos Noguera
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
Email: cnoguera@soft.vub.ac.be

Viviane Jonckers
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
Email: vejoncke@soft.vub.ac.be

***Abstract*—Traceability links are used to ensure co-evolution among related software artefacts. That is, to ensure that changes to the application are correctly and completely propagated. Much emphasis has been put on reverse engineering and co-evolution of vertical traceability links (i.e., across artefacts of the different levels of abstraction like UML, requirements documents, source code). However, today's applications require automatic support to deal with complex dependencies across heterogeneous source code entities (e.g., it is not uncommon for web applications to require multi-language and multi-paradigm programming). This paper introduces MaTraCa, an Eclipse plugin to maintain traceability links among entities of the same level of abstraction (i.e., code) in complex applications. MaTraCa stands for Managing Traceability Changes. MaTraCa focuses on horizontal links, that is, relations across software artefacts of the same level of abstraction such as source code, config files, html forms, etc.**

I. Introduction

The use of traceability links has been motivated mostly by the need of co-evolving interdependent software artifacts. However, an often neglected traceability concern is managing the complexity of current applications where the heterogeneity of source code artifacts and the lack of support of IDEs hinder source code development and evolution tasks. In consequence, we propose MaTraCa to validate traceability links between heterogeneous source code artifacts.

MaTraCa is designed to be independent of the type of link analyzed. However, the current prototype can only handle links between entities/elements of two types of source code artifacts: source code (Java) and XML. We chose XML artifacts because XML is a common way to describe meta-data or configuration of applications, however current IDEs lack of support to maintain the links of XML entities to other source code resources. In fact, several popular APIs and libraries can be configured using XML. For instance, Spring, Maven, and Hibernate.

The main contributions of MaTraCa's tracing and validation facilities are:

- An API to identify relevant XML nodes and their characteristics
- A declarative means to document heterogeneous traceability links
- An implementation that allows the maintenance of heterogeneous traceability links: which tells the developer which links should be updated based on the owner of the dependency.

In the remainder of this paper, we present a summary of approaches tackling traceability links in section II, then we provide an overview of MaTraCa's tracing and validation facilities in section III, after that we summarize key implementation details in section IV, next we explain why the tool is extensible to any link in section V. Finally, the current state of the tool and the future work are summarized in section VI.

II. Related work

Much work on traceability links has focused on the automatic recovery of traceability links (i.e., vertical traceability). Many of these approaches use information retrieval techniques [1] which allows them to handle traceability links across heterogeneous software artifacts. However, empirical results have shown that obtaining fine-grained traceability links does not necessarily outweigh the cost of extracting them [2]. Finally, it has been found that vertical traceability links do not tend to propagate their changes bottom-up (i.e., changes in code are unlikely to affect the design) [3].

There are also techniques to recover horizontal links, like logical coupling [4]. Logical coupling mines for possible links between source code entities that tend to be modified at the same time. However, logical couplings requires previous changes among the related entities *at the same time* and cannot provide any rationale for the mined relations. In contrast, to our tool relies only on the source code artifacts and on a meta-level description of entities that should be linked (i.e., there is certainty that the relation should exist and its rationale is known).

We found three other approaches based on the definition of the links. First, the implementation of traceability links as queries in a database that enable the validation of design and requirement constraints over source code [5]. Second, using a meta-model to link source code, UML, and unit tests, plus adding manually propagation rules to identify whenever a change in any of these artifacts would impact any of its related artifacts [6]. And third, converting heterogeneous source code artefacts to a GraphML representation (via XML) and specifying the links with XML to have a semiautomatic change impact detection [7]. However, the last one did not provide much details, for instance, it is not clear whether the links are defined and enforced at the meta-level or at the instance-level or whether the change impact detection requires first change detection via the version control system.

III. MaTraCa: Managing Traceability Changes

This section demonstrates the usefulness of MaTraCa via a running example[1]. Suppose XML files containing the definition of error messages (see listing 1), and that the usage of these definitions is done via calls to the "message" method of the "MessageManager" class (see listing 2)[2].

Listing 1. Example of error message definition

```
<message id="MSG000001">
        <severity>20</severity>
        <parameters>
                <parameter name="message" type="String" />
                <parameter name="query" type="String" />
                <parameter name="timing" type="Long" />
        </parameters>
</message>
```

Listing 2. Example of usage of error message defined in the XML file

```
MessageManager m = new MessageManager();
...
m.message("MSG000001",
        new Object[] {"Search",
                "Some query ...",
                Long.valueOf(100)},
        null);
```

In this case, the traceability link is defined between the Java code that logs the error message, and the XML files that declare the messages (see Fig. 1). Regardless of the origin of traceability links (i.e., reverse engineering or manual documentation), it is necessary to express them in an unambiguous and explicit way. A traceability link describes an equivalence relation between '*implementation concerns*'. The equivalence relation indicates that they require consistent updates or co-evolution. Therefore, in theory, it is enough to describe the source and target implementation concerns that require co-evolution to have traceability links. An implementation concern is defined as an enumeration of the attributes that uniquely identify those entities which require similar co-evolution.

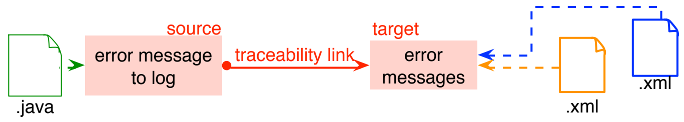


Fig. 1. Traceability links from logging code to definition of error messages.

The goal behind our traceability maintenance facilities is to provide 'on-the-fly' development support. In this case, a link should look as broken when a message ID is used in code that doesn't exist in messages.xml (e.g. MSG000015), if the message exists but is not used correctly, then its parameters would show that there is not a match between the call and the way it is defined. Our framework provides the following validation steps for traceability links (see Fig. 2a):

1) Detection of entities that should comply with a traceability link (see Fig. 2b)
2) Configuration of valid/invalid or relevant/irrelevant links (done via double click and signaled as a colored/gray icon in the left most column of each link –see Fig. 2b).
3) Identification of broken links (i.e., links for which there is only source or target implementation concern but not known match –see Fig. 2b)
4) Identification of links that have changed (see Fig. 2g and Fig. 2h).

We have also identified the following usability requirements:

Navigation: Being able to click on the resulting software artifact and open the default editor in the line of corresponding to that element/entity (see Fig. 2d),

Parameters: Taking into account parameters[3] in the traceability link (see Fig. 2e and Fig. 2f),

Search: Being able to find a particular result by name of any of the elements/entities that conform the traceability link (see Fig. 2e).

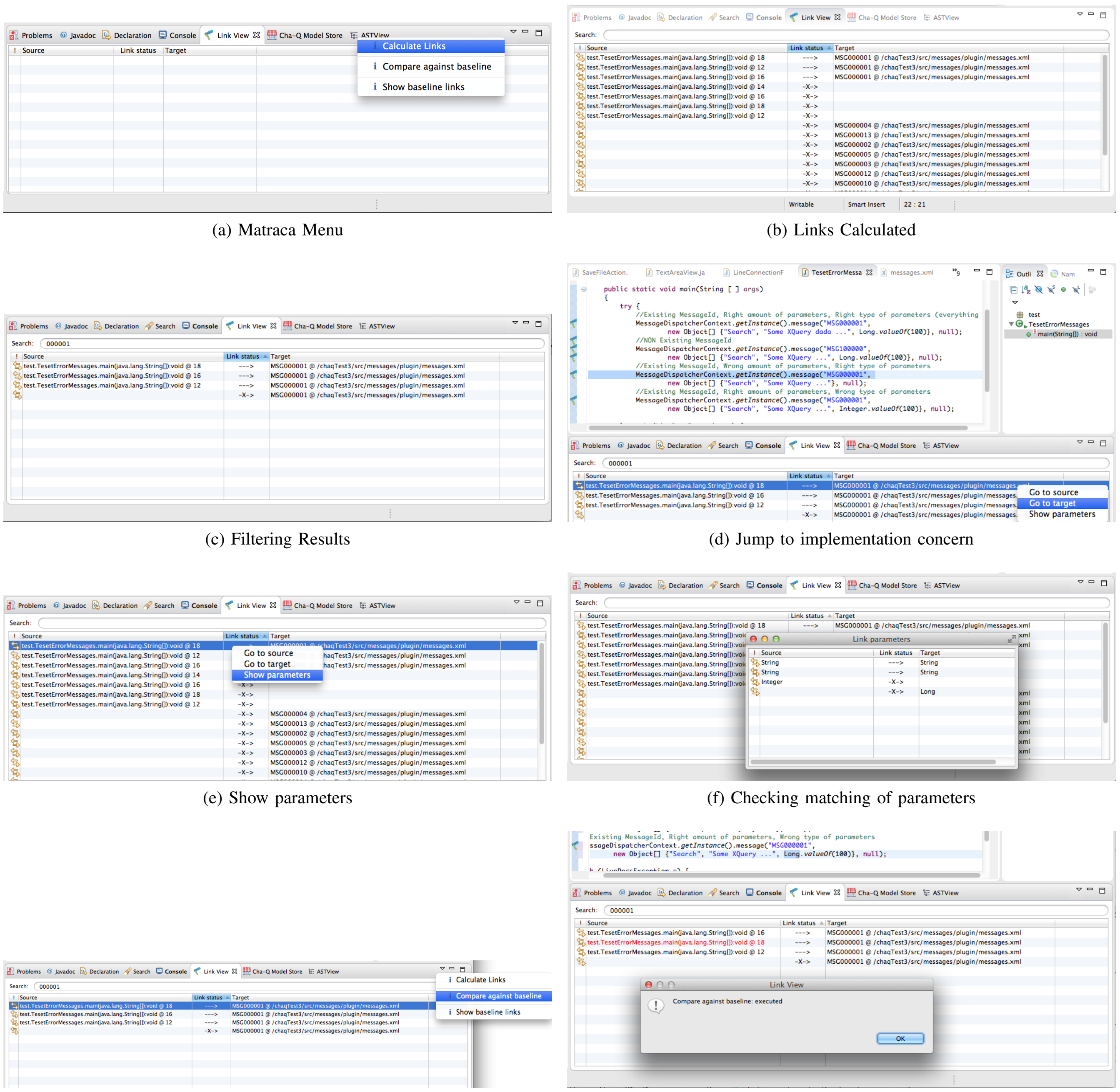
IV. MaTraCa's design choices

MaTraCa is implemented on top of the the CHAQ meta model [8]. This meta model offers to MaTraCa an AST representation of the Eclipse projects chosen to be analyzed (see Fig. 3). The representation of AST nodes is complemented with bindings that provide the equivalent FAMIX [9] meta model abstractions, while each FAMIX entity has a link to its corresponding AST node. Similarly, the model offers the parsed representation of each XML file in the projects. A layer of basic Clojure predicates permits basic reasoning about these (AST and parsed XML trees). These predicates are used to filter the entities that are relevant for MaTraCa (i.e., source or target implementation concerns). *Therefore, MaTraCa is not limited to a single definition of links but allows to describe any source code entity in the model as source or target of the links.* In fact, the two examples mentioned in the paper are two different configurations of MaTraCa. MaTraCa's plugin builds traceability links from source and target implementation concerns. These concerns are located via logic predicates that state what identifies source (domain) or target (co-domain) implementation concerns (see Fig. 4). The predicates that define the implementation concerns must return a list of Java objects whose first element is a string identifier of the implementation concern (shown in the user interface), the value that is used to establish a match between two implementation concerns, and the location of each implementation concern (e.g., file, line, and sometimes starting character). This allows MaTraCa to create implementation concerns from the results of the domain and co-domain predicates, which then are matched and used to define links.

[1]This example is a simplification of one of the use cases for MaTraCa provided by one of the industrial partners of the CHAQ project

[2]The method "message" receives three parameters: the identifier of the error (as a String), the parameters of the error message (as an array of Objects), and the cause of the error message (as a Throwable).

[3]Notice that some traceability links (like the one shown in the example) can have parameters. This means that although the basic matching is done via some characteristics (e.g., the error id), the link itself can have further matching constraints (e.g., the types of attributes that the error message requires). Therefore, the parameters of a traceability link are defined as another traceability link.



(a) Matraca Menu

!	Source	Link status	Target
	test.TesetErrorMessages.main(java.lang.String[]):void @ 18	--->	MSG000001 @ /chaqTest3/src/messages/plugin/messages.xml
	test.TesetErrorMessages.main(java.lang.String[]):void @ 12	--->	MSG000001 @ /chaqTest3/src/messages/plugin/messages.xml
	test.TesetErrorMessages.main(java.lang.String[]):void @ 16	--->	MSG000001 @ /chaqTest3/src/messages/plugin/messages.xml
	test.TesetErrorMessages.main(java.lang.String[]):void @ 14	-X->	
	test.TesetErrorMessages.main(java.lang.String[]):void @ 16	-X->	
	test.TesetErrorMessages.main(java.lang.String[]):void @ 18	-X->	
	test.TesetErrorMessages.main(java.lang.String[]):void @ 12	-X->	
		-X->	MSG000004 @ /chaqTest3/src/messages/plugin/messages.xml
		-X->	MSG000013 @ /chaqTest3/src/messages/plugin/messages.xml
		-X->	MSG000002 @ /chaqTest3/src/messages/plugin/messages.xml
		-X->	MSG000005 @ /chaqTest3/src/messages/plugin/messages.xml
		-X->	MSG000003 @ /chaqTest3/src/messages/plugin/messages.xml
		-X->	MSG000012 @ /chaqTest3/src/messages/plugin/messages.xml
		-X->	MSG000010 @ /chaqTest3/src/messages/plugin/messages.xml

(b) Links Calculated

Search: 000001

!	Source	Link status	Target
	test.TesetErrorMessages.main(java.lang.String[]):void @ 18	--->	MSG000001 @ /chaqTest3/src/messages/plugin/messages.xml
	test.TesetErrorMessages.main(java.lang.String[]):void @ 16	--->	MSG000001 @ /chaqTest3/src/messages/plugin/messages.xml
	test.TesetErrorMessages.main(java.lang.String[]):void @ 12	--->	MSG000001 @ /chaqTest3/src/messages/plugin/messages.xml
		-X->	MSG000001 @ /chaqTest3/src/messages/plugin/messages.xml

(c) Filtering Results

```
public static void main(String [ ] args)
{
    try {
        //Existing MessageId, Right amount of parameters, Right type of parameters (everything
        MessageDispatcherContext.getInstance().message("MSG000001",
                new Object[] {"Search", "Some XQuery dada ...", Long.valueOf(100)}, null);
        //NON Existing MessageId
        MessageDispatcherContext.getInstance().message("MSG100000",
                new Object[] {"Search", "Some XQuery ...", Long.valueOf(100)}, null);
        //Existing MessageId, Wrong amount of parameters, Right type of parameters
        MessageDispatcherContext.getInstance().message("MSG000001",
                new Object[] {"Search", "Some XQuery ..."}, null);
        //Existing MessageId, Right amount of parameters, Wrong type of parameters
        MessageDispatcherContext.getInstance().message("MSG000001",
                new Object[] {"Search", "Some XQuery ...", Integer.valueOf(100)}, null);
```

(d) Jump to implementation concern

(e) Show parameters

Link parameters

!	Source	Link status	Target
	String	--->	String
	String	--->	String
	Integer	-X->	
		-X->	Long

(f) Checking matching of parameters

(g) Compare against baseline

```
Existing MessageId, Right amount of parameters, Wrong type of parameters
ssageDispatcherContext.getInstance().message("MSG000001",
      new Object[] {"Search", "Some XQuery ...", Long.valueOf(100)}, null);
```

Search: 000001

!	Source	Link status	Target
	test.TesetErrorMessages.main(java.lang.String[]):void @ 16	--->	MSG000001 @ /chaqTest3/src/messages/plugin/messages.xml
	test.TesetErrorMessages.main(java.lang.String[]):void @ 18	--->	MSG000001 @ /chaqTest3/src/messages/plugin/messages.xml
	test.TesetErrorMessages.main(java.lang.String[]):void @ 12	--->	MSG000001 @ /chaqTest3/src/messages/plugin/messages.xml
		-X->	MSG000001 @ /chaqTest3/src/messages/plugin/messages.xml

Link View

Compare against baseline: executed

OK

(h) Show changes in baseline links

Fig. 2. Managing the traceability links with MaTraCa.

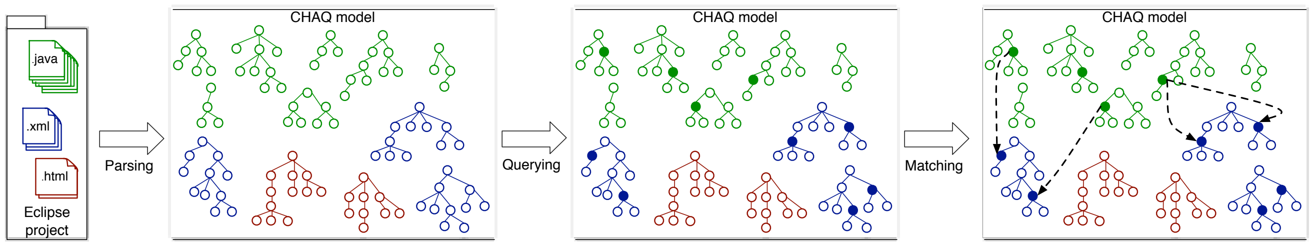


Fig. 3. Process followed by MaTraCa to detect and validate links.

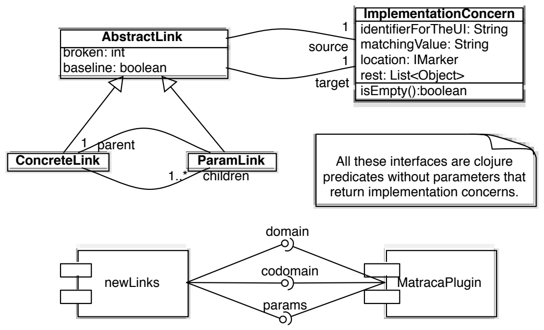


Fig. 4. MaTraCa's high level design.

V. MaTraCa's extensibility

Current web-based application require several technologies. Therefore, web applications are an ideal case study to check horizontal traceability links over heterogeneous source code artifacts.

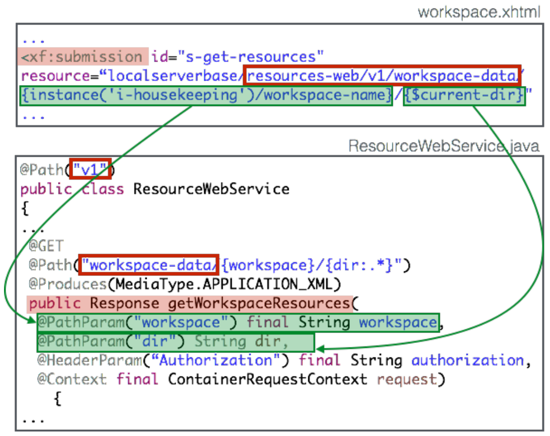
In order to evaluate MaTraCa's extensibility we described the traceability links in the Java API for RESTful[4] Web Services (JAX-RS) i.e., (JSR 311). That is, the source of our links would be the URLs defined in the web page describes HTTP operations (e.g., get, put, post, delete, etc.) on domain entities using XForms[5], while their target are Java method that implement the corresponding web-service. A large amount of jumps is required to check that the resource mentioned in the URI of a request corresponds to the right method in the server. Furthermore, these URIs are dynamically constructed depending on the state transitions that the client has made, so, checking them becomes time consuming and error prone.

In this case, the traceability links start from the submission resource of the XForm to the Java method that should be called when a given web service is chosen (see Fig. 5). The *domain* was defined the XForm-submission resource in xhtml files [6]. The *codomain* was defined as the concatenation of the location of web-services, then it looks for the class that implements the web application (defined in an XML file) which contains the simple name of the class that implements the web service called by the html page, once that file is located it concatenates the value of the @Path annotation at the class and method level. A video of this version of MaTraCa can be found at https://sites.google.com/site/alozanoresearch/demos.

[4]REST stands for Representational State Transfer. REST is an alternative to SOAP and WSDL based Web services.

[5]XForms is an XML way of defining web interfaces. XForms is inspired on the MVC pattern. In XForms, the controllers are the actions triggered by updating the model or the views form. These can be identified by action definition nodes (e.g., *<xf:varName ref=...>, <xxf:varName submission=...>, <bind ...>, <submission action=...>*). The view correspond to the HTML elements that provide the presentation of the document and the XPath associated to them. Finally, the model is the data to be edited and submitted which is described inside the *<model>* tag.

[6]Notice that the submission resource can refer to variables defined in the XForm (i.e., the XML is dynamically built), and therefore MaTraCa cannot rely on string matching across entities.



```
...
<xf:submission id="s-get-resources"
resource="localserverbase/resources-web/v1/workspace-data/
{instance('i-housekeeping')/workspace-name}/{$current-dir}"
...
```

```
@Path("v1")
public class ResourceWebService
{
...
 @GET
 @Path("workspace-data/{workspace}/{dir:.*}")
 @Produces(MediaType.APPLICATION_XML)
 public Response getWorkspaceResources(
  @PathParam("workspace") final String workspace,
  @PathParam("dir") String dir,
  @HeaderParam("Authorization") final String authorization,
  @Context final ContainerRequestContext request)
    {
...
```

Fig. 5. MaTraCa's traceability links from XForms to Java. Matching elements are shown filled light red boxes, matching criteria with empty red-bordered boxes, and parameters with filled light green boxes.

VI. Conclusion

This paper presents MaTraCa, a tool that allows the documentation of horizontal traceability links, the establishment a baseline (focus in those links of interest to the developer), the evaluation of broken/obeyed links, and the comparison of the current version of the code against the baseline. We have shown the usefulness of our prototype by means of a simple example (error messages) and its extensibility with another example (XForms submissions and the Java implementation of RESTful web services). Currently we are preparing in a pilot case with an industrial partner (using the traceability of REST links) to evaluate the impact of the tool on the maintenance of traceability links.

References

[1] M. Borg, P. Runeson, and A. Ardö, "Recovering from a decade: A systematic mapping of information retrieval approaches to software traceability," *Empirical Softw. Engg.*, vol. 19, no. 6, pp. 1565–1616, 2014.

[2] A. Egyed, F. Graf, and P. Grunbacher, "Effort and quality of recovering requirements-to-code traces: Two exploratory experiments," in *Proc. of the IEEE Int'l Requirements Engineering Conf. (RE)*, 2010, pp. 221–230.

[3] M. Hammad, M. L. Collard, and J. I. Maletic, "Automatically identifying changes that impact code-to-design traceability during evolution," *Software Quality Journal*, vol. 19, no. 1, pp. 35–64, Mar. 2011.

[4] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proc. of the International Conference on Software Maintenance (ICSM)*, 1998, pp. 190–198.

[5] S. P. Reiss, "Incremental maintenance of software artifacts," in *Proc. of the IEEE Int'l Conf. on Software Maintenance (ICSM)*, 2005, pp. 113–122.

[6] S. Lehnert, Q.-u.-a. Farooq, and M. Riebisch, "Rule-based impact analysis for heterogeneous software artifacts," in *Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR)*, 2013, pp. 209–218.

[7] I. Pete and D. Balasubramaniam, "Handling the differential evolution of software artefacts: A framework for consistency management," in *Doctoral Symposium of Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 599–600.

[8] C. D. Roover, C. Scholliers, V. Jonckers, J. Pérez, A. Murgia, and S. Demeyer, "The implementation of the CHA-Q meta-model: A comprehensive, change-centric software representation," *ECEASST*, vol. 65, 2014.

[9] S. Ducasse, J. Laval, U. B. Nicolas Anquetil, A. Hora, and T. Girba, "MSE and FAMIX: an interexchange format and source code model family," INRIA LNE- LIRMM, Tech. Rep. hal-00646884,, November 2011.