Automatic Classification of Domain Constraints for Rich Client Development

Manuel Quintela-Pumares, Daniel Fernández-Lanvin, Alberto-Manuel Fernández-Álvarez, Raúl Izquierdo
Computer Sciences Department
University of Oviedo
Oviedo, Spain
e-mail: manuel.quintela@gmail.com, dflanvin@uniovi.es, alb@uniovi.es, raul@uniovi.es

***Abstract*— The current trend in web development, powered by the popularization of technologies like Ajax or platforms like iOS and Android, leads developers to gradually leave the classic *light-weight* web client in favor of *rich clients.* These clients manage not only presentation logic, but also business rules that transform part of the domain model that afterwards must be reintegrated in the server. This temporary duplication and transformation of part of the domain model force developers to deal with the management of the domain constraints that must be retrieved and applied in the client. This is a complicated and error prone task that usually involves redundant design and implementation on both sides. This work describes a tool that, given a domain model with its complete set of constraints, and the subset of classes that are required in the client, automatically identifies those constraints that the client requires and that can be applied separately from the server, classifying them according to their level of dependency with the server.**

Keywords-rich clients; constraints; OCL; UML.

I. INTRODUCTION

The architecture of web applications has been continuously evolving since the popularization of the primitive transactional script based systems. The current trend, powered by the popularization of technologies like Ajax or mobile platforms where native applications connect to the internet, leads developers to gradually leave the classic web *light-weight* client model, in which the client deals mostly with presentation logic [1], to a more distributed model, in which a Subset of the Domain Model (SDM) is retrieved and transformed in the client, to be redelivered back to the server to be reintegrated with the complete domain model (CDM) located on the server [1][2][3]. Some well-known web applications, like Google Docs or Google Calendar, are good examples of this approach.

This rich Internet application (RIA) architectural model carries a better user experience, since the classical delay between requests is mitigated [4]. However, it also involves important issues during the design and implementation [5]. The temporary splitting of the domain model, and its later reintegration in a multi user environment, force developers to figure out which of the constraints of the model should be checked in the client [6], whether they should be transformed, and which and how they should be checked again once the transformed sub-domain is reintegrated in the server [7].

Identifying at design time the constraints that can be safely verified on the client is a tricky job, and finding out if the existing ones can be modified -so that they can be located on the client- is a complicated and error prone task. Even when some constraints could be fully checked on the client, a redundant checking must be done back in the server for security reasons [8], requiring a redundant implementation.

Also, if there are different teams working at client and server side, human coordination problems can lead to inconsistencies. This problem is aggravated by the ever present changes in the requirements, making the constraints variable in both client and server. All these elements make the design and implementation of constraints a very complex, tedious and error prone task, especially as requirement changes accumulate over time [8].

All these problems would be avoided if we could automatically determine which of the constraints can be checked in the client and which cannot, and how they should be managed all along the process. This would support de dynamic generation of the control logic that manages those constraints in the client, avoiding redundant implementation and turning the development process more agile.

In our understanding, all the information we need for that can be deduced, for a specific SDM, from the information contained in the CDM in terms of entities, relationships and constraints.

To address these problems, we have designed a tool that can aid developers to easily produce the client subset using the CDM, its UML (Unified Modeling Language) [9] class diagrams and OCL (Object Constraint Language) [10] constraints as input parameters. A new class model will be generated for the client, maintaining the relations according to that subset, and discarding all unrelated classes, relations, methods and constraints. Since some of the constraints will require information from the server to be checked, and involve different levels of coupling, the tool automatically identifies and classifies the constraints that are relevant to the client by their dependency degree: (a) Completely independent of the server, (b) Can be dependent to the server in some circumstances and (c) Completely dependent to the server.

The rest of this paper is organized as follows. Section II describes the method we propose for the automatic classification of constraints. Section III provides an example illustrating how the tool works. Section IV addresses the related work. Section V presents the conclusion and future work. The acknowledgement closes the article.

II. Method for the automatic classification of constraints

We propose a method where the designer creates the UML model for the CDM located on the server, including its constraints described in OCL, as he/she would usually do, and then determine the classes and interfaces from the server model that corresponds to the SDM. With this information, a new class model and a new set of constraints are generated for the client. The constraints for the client are analyzed and automatically classified according to their level of dependency with the server, detecting those that may be problematic and require special attention.

The tool we have developed for implementing this method is a programmatic API written in Java that automatically generates the SDM from the CDM. Its implementation is based on EMF Ecore [11] class models and OCL files [10]. The input is the Ecore and OCL files that describe the server model, and the classes that belong to the SDM. The output will be a new Ecore file with the class model for the client, a new OCL file with the constraints that can be checked on the client, and an additional text file with information about the modifications of the class diagram, and the analysis, classification and documentation about the constraints.

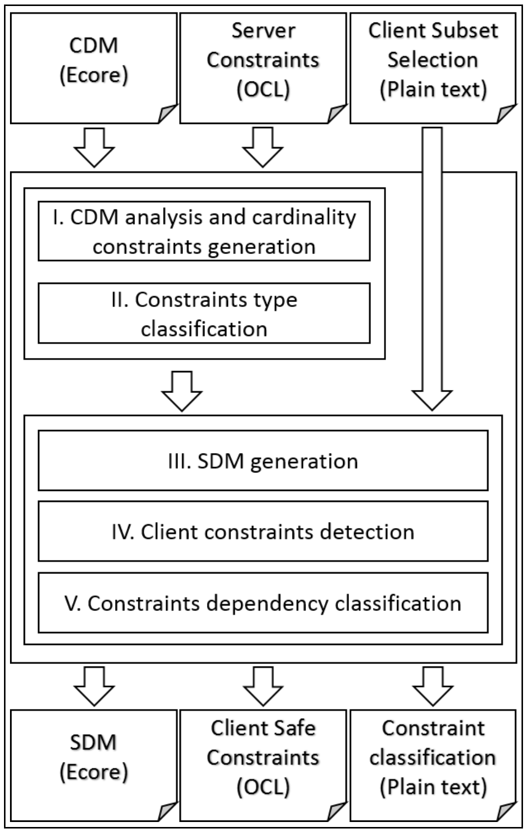
CDM (Ecore)
Server Constraints (OCL)
Client Subset Selection (Plain text)
I. CDM analysis and cardinality constraints generation
II. Constraints type classification
III. SDM generation
IV. Client constraints detection
V. Constraints dependency classification
SDM (Ecore)
Client Safe Constraints (OCL)
Constraint classification (Plain text)

Figure 1. Inputs, outputs and processes that the tool carries out.

A. *Analizyng the CDM and its constraints*

The tool first analyzes the classes in the model, their attributes, methods and relationships. To ease the analysis of the cardinality constraints described in the class model relationships, those constraints are automatically transformed to OCL language, so that they can be processed homogenously with the rest (Figure 1, I.).

For every constraint, it collects information about the classes that are being referenced in its body, as well as the attributes that are being referenced and their primitive types, or the return type and parameters that are being used from their methods.

With all this information, the constraints are classified (Figure 1, II.) applying the following criteria:

- *Attribute constraint*: A constraint that only concerns a single attribute of the context class. We deduct this by observing the parameters that receive the operations of the constraint. If it contains a single property call whose type is of a primitive type, it is classified as an attribute constraint.
- *Object constraint*: A constraint that concerns more than one attribute of the context class. We deduct this as we did with the attribute constraint. If it contains different property references whose types are primitive types, it is an object constraint.
- *Class constraint*: A constraint that concerns several instances of the context class, and not elements of any other class. We determine this by observing if the types of the references (navigations, property accesses or method invocations) or parameter calls correspond to the context class, and not any other classes.
- *Domain constraint*: A constraint that makes reference in its operation to elements of other classes different than the class of its context. We calculate this in the same way that class constraints, but if a class has a different type than the context class, it is a domain constraint.

B. *Generating the SDM for the client*

After analyzing the CDM, the tool uses the subset of classes that the designer has selected to generate the SDM (Figure 1, III.). The new class model will contain only the classes described in the client subset. The relationships affecting the SDM classes are maintained in the new model. Those that connect any of those SDM classes to any class outside the SDM are processed as follows:

- Association, aggregation and uses relationships: If a class within the client subset has any of these types of relationship with a class outside the client subset, the relationships and the classes outside the client subset will be removed from the SDM.
- Inheritance relationships: A parent class can exist without its child classes, but in a class model a child class does not make sense without its parent classes. To address this problem, if a child class is included in the client subset by the designer, the tool automatically

includes its parent class. If there are various levels of inheritance above the selected child class, all the inheritance hierarchy for that class will be recursively included in the client subset.

- Interface relationships: We take the same approach as in inheritance relationships.
- Composition relationships: Composition is a relationship that models a strong relationship between a component and a container class, tying their lifecycles tightly. We consider that a component class can make sense without its relationship with a container class, but not the other way around. If the client subset includes a container class, we automatically include also its component classes and their composition relationships even if the designer did not consider them for the client subset. As in inheritance relationships, if the automatically included component classes are also containers of other classes, their components will be recursively included in the client subset.
- Methods: If the classes included in the client subset contain methods whose signature contains classes outside the client subset, those methods will be deleted from the class. We consider that, if those classes are kept outside the client subset, the methods that make reference to them will not be needed on the client.

C. *Selecting and classifying the constraints for the client*

The tool will select the OCL constraints whose context matches the elements in the client subset. The rest of the constraints will be discarded for the client (Figure 1, IV.). Constraints whose context is not in the SDM will not be considered due to the fact that there will not be any object of those classes in the SDM object graph.

The tool will also warn the designer about the level of dependency of each constraint with the server (Figure 1, V.). We define three levels of dependency:

1. *Completely independent*: All *attribute* and *object* constraints are completely safe for being checked independently on the client, since all the elements needed to check those constraints are already within the SDM object graph.
2. *Potentially dependent*:
   a. *Class* constraints may or may not be checked safely within the client. This will depend upon how the behavior of the client objects is defined. If every object of that class is always on the client, the constraint will be always safe. If the objects are requested from the server under request, the constraint could not be safe without some previous communication with the server in order to retrieve the required objects.
   b. *Domain* constraints that exclusively make reference to classes within the client subset are in the same circumstances as the *class* constraints. Their safety depends on the way the model is being managed. If a constraint needs information from objects that are not currently on the client, communication with the server will be required.
3. *Completely dependent*: *Domain* constraints that make reference to classes that are not in the client subset will always be dependent from the server, since they reference elements that are not considered on the client. These constraints should be delegated to the server, or when possible, be reformulated by the designer so that at least part of their operations can be checked on the client, delegating the rest to the server.

The output of this whole process is an Ecore file with the resulting SDM, a text file with the results of the modifications made from the CDM, and the analysis of the constraints related to the client and their classification. It also generates an OCL file containing the constraints that can be checked on the client without modification (those classified as *completely independent* or *potentially dependent*), and excluding the *completely dependent* (they cannot be checked on the client without modification).

III. THE ROYAL AND LOYAL EXAMPLE

The Royal and Loyal model [10] is a popular example usually used to explain the OCL language. We used a version of it to show the way the tool works if we need to develop a rich client for managing the addition of new Loyalty Programs. Figure 2 shows the Ecore model of the CDM located on the server, simplified for displaying only class names and references.

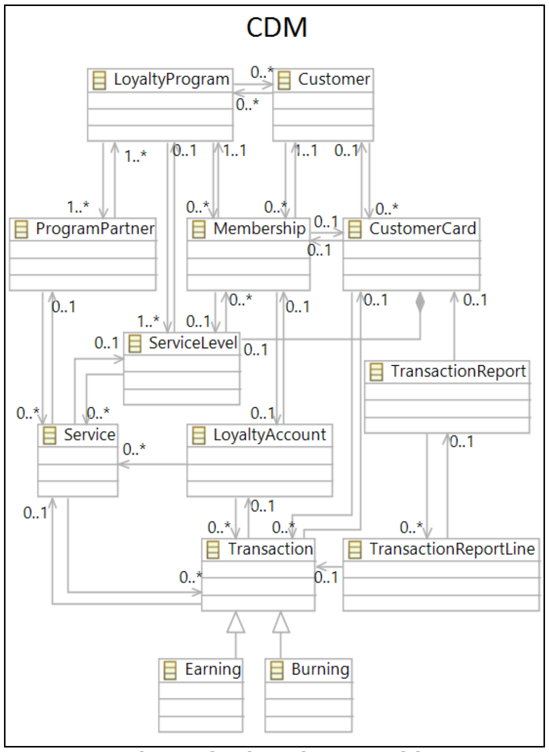

Figure 2. The Royal and Loyal Ecore model as CDM.

The version we used for this example has 22 constraints defined. After being analyzed by the tool, it automatically generates 28 additional constraints based on the cardinalities of the relationships of the class model, resulting a total of 50 constraints to process.

The model in our rich client will have the following classes from the CDM located on the server: "*Service*", "*ServiceLevel*", "*LoyaltyProgram*" and "*ProgramPartner*". Those classes will allow us to define new Loyalty Programs, partners, and the services they provide. The other functionalities that the full model provides, such as defining customers or managing their subscriptions to loyalty programs are out of the scope of this client.

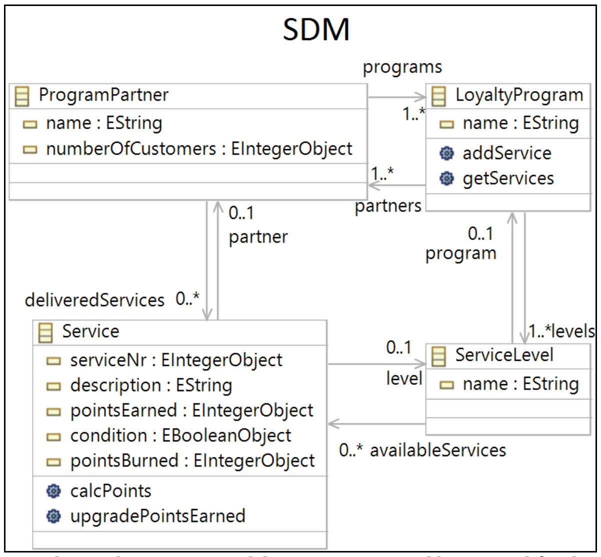

Figure 3. The resulting Ecore model as SDM generated by our tool for the client.

In Figure 3, we show the resulting SDM. Figure 4 presents the information it provides about the methods that have been deleted from the original class (due to their dependence from elements outside the client model), and also about the relationships that have been deleted from the original model.

```
Deleted classes:
Transaction,    Customer,    CustomerCard,    Membership,
LoyaltyAccount,  Burning,  Earning,  Transaction  Report,
TransactionReportLine
-----------------------------------------------
Deleted Methods:
Customer->        enroll,        selectPopularPartners,
enrollAndCreateCustomer, addTransaction, getServices
-----------------------------------------------
Deleted relationships:
Service ->Transaction: transactions
LoyaltyProgram -> Membership: memberships
LoyaltyProgram -> Customer: participants
ServiceLevel -> Membership: membership
```

Figure 4. The tool generates information about the classes, methods and relationships that are deleted in the process.

Regarding the constraints, it generates a plain text file describing those that affect each class, classifies them, and points out if they can be checked on the client or not. It detects 14 related to this SDM, 13 of them are classified as *domain* constraints and 1 as *attribute* constraint. After analyzing the dependency of these constraints, 1 is detected as *completely independent*, 9 as *potentially dependent*, and 4 as *completely dependent*. Figure 5 shows one constraint of each level of dependency as an example.

```
context Service::upgradePointsEarned(amount : Integer)
post   postServiceUpgradePointsEarned:   calcPoints()   =
calcPoints@pre() + amount
        Classification: attribute
        Context Class: Service
        Referenced Classes: []
        Classes in context operation: [Service]
        Dependency: Completely independent
-----------------------------------------------
context LoyaltyProgram  inv firstLevel:
levels->first().name = 'Silver'
        Classification: domain
        Context Class: LoyaltyProgram
        Referenced Classes: [ServiceLevel]
        Dependency: Potentially dependent
-----------------------------------------------
context ProgramPartner  inv totalPoints:
deliveredServices.transactions.points->sum() < 10000
        Classification: domain
        Context Class: ProgramPartner
        Referenced Classes: [Service, Transaction]
        Dependency: Completely dependent
```

Figure 5. A selection of three of the resulting constraints, each one with a different level of dependency.

There are some constraints that can always be checked on the client without communicating with the server, like the postcondition for "*upgradePointsEarned*".

Some of the constraints have all the elements needed for checking the constraint in the client model, but it may need to communicate with the server to update the data, like the "*firstLevel*" invariant.

Other constraints reference elements outside the client model, that is, objects of that class don't exist on the client, like the "*totalPoints*" invariant.

The problem of having constraints on the client that reference elements that only exists on the server can be solved in several ways. The most straightforward way would be delegating the checking to the server. However, if we still want to make the checking on the client, it can be achieved by adding some kind of proxy that requests from the server the dependent values needed to check that constraint.

Finally, it creates an OCL file with the constraints that are *completely independent*, and *potentially dependent* (10 in total). It excludes the *completely dependent* ones since they refer to classes that are not on the SDM. The user should use this information to figure out the best way to adapt those dependent constraints for the SDM.

IV. Related work

There are several proposals that encourage locating more responsibilities on the client side rather than delegating them to the server. Hallé and Villemaire's [12] proposal is centered on rich clients that connect with web services, a system that checks the preconditions defined on the service interface on the client side before making the request, avoiding an unnecessary expenditure of resources on the server. Heidegger and Thiemann [13] add annotation-like pre and postcondition support to Javascript, a language widely used to develop rich clients where developing complex business logic is more common every day. The work presented by Zhang [14] suggests to move all business logic to the client, and leaving on the server only a database accessible through REST services. Leff and Rayfield [15] show a client designed to work in mobile environments where connection can be lost, defining mechanisms that support offline functionality and maintaining the integrity when the client is back online. All these proposals recognize the benefits of moving tasks to the client side and try to address some aspects of making integrity checks on clients. However, in all of them the responsibility of deciding which constraints are relevant on the client must be manually done by the developer.

Other authors try to solve the implementation problems of having constraints on a rich client [16][17], since popular tools to address business rules and validations are still very limited on this scenario. Rule engines, like Drools [18] or ILog [19], are a suitable solution for the server side, but they are not designed to deal with the ones located on the client. Tools to address client side validations like Struts [20], jQuery Validation Plugin [21], or Simfatic [22] are still limited to simple form checking, but are not designed to cover the complexity that client side business rules can require.

Liang et al. [16] propose a system in which validations are defined on an XML file, managing constraints that involve a combination of several attributes on the client's forms. This automates the implementation of part of the client side constraints, and improves the maintenance process. However, they explicitly left out of their scope the more complex and problematic *class* and *domain*. Schmidt et al. [17] designed a rule engine for the client side based on the RETE algorithm, where the constraints are defined on a file on the server. While they support the definition of complex constraints and even their delegation of to the server, the specific constraints affecting the client have to be manually specified. Most of these solutions would benefit with our proposal.

Louwsma et al. [7] analyzes the problems derived from managing constraints in a rich client for a GIS, where the user can add elements to the map over a graphic interface that will be updated to a central database. They propose a framework based on UML and OCL for the specification of constraints, and suggest several constraint classification criteria, but their implementation is hard-coded and delegates all the constraint checking to the database. They identify the problem of having constraints that can affect both client and server, proposing as future work that some types of constraints should be validated on the client for a better user experience, as well as automatic classification and detection of conflicting constraints, and their automatic implementation from a central specification.

Other previous works specifically address the problem of deciding how to split applications between different machines in an automatic way. Proposals like J-Orchestra [23] or Coign [24] process existing compiled applications, analyzing the way their different elements communicate. By means of code instrumentalization, they provide stubs to allow the division in different parts that can communicate, maintaining the same functionality. Also, Yang et al. designed a platform based on the Hilda language [25] with a runtime in both client and server that decides dynamically which elements of the application should run on the client and which on the server, basing on the characteristics of the client device.

All these approaches use different strategies to decide which the optimal distribution of their components is, by gathering information about the application behavior (like communication delay between elements, the size of the data transmitted, memory usage, capacity of the devices, or the demand by users of a certain functionality). However, none of these proposals deals with the problem of constraint redistribution. They add proxies to communicate the different split elements of the original design but do not change them to support constraint checking in order to maximize UI usability and responsiveness. All these solutions could benefit from automatic constraint classification and modification techniques in those cases in which client responsiveness is a priority.

Outside the scope of rich client development, techniques for automatically adapting OCL constraints have been developed to fit different purposes. Hassam et al. [26] propose techniques for automatically maintaining the consistency of the OCL constraints after applying modifications to the UML model. For each change made to a model, their tool identifies the OCL constraints affected by it, and then decides if the constraints have to be removed because they are no longer relevant, of if they can be automatically modified to be consistent with the modified model. Cabot and Teniente [27] developed techniques for automatically modifying constraints and domain models to achieve a more efficient integrity checking. For doing that, they develop techniques for simplifying OCL constraints, identify which operations trigger certain OCL constraints, and reformulate the constraints in the most efficient way given the possible operations found in the model.

These proposals acknowledge the problem of delegating to the designer the task of revising existing OCL constraints for achieving certain objectives when that tasks can be deduced from the UML model. In addition to this, although they are designed to solve scenarios different than the one we propose, the principles behind the identification of which OCL constraints need attention, and some of the automatic modification mechanisms described in them could be useful for future developments of our tool.

V. CONCLUSION AND FUTURE WORK

The proposed tool deals with the generation of the new domain model for the client, selecting and classifying the constraints for the client, and automatically identifying the conflicting elements of the constraints that are not completely independent from the server. At its current state, it removes from the designer the responsibility of modeling the part of the client class model that overlaps with the server, providing useful documentation about the constraints that potentially affects the client.

If the designer wants to make a domain model on the client where as many as possible validations are made locally, the tool can help him/her to make better informed decisions while trying to modify the constraints and the client model to fit that purpose.

This approach can also complement the existing tools that deal with the implementation of constraints on the client, but currently delegate to developers the responsibility of organizing them.

We have previously developed means to achieve automatic error recovery in rich clients [28], letting the developer to choose which parts of the model require this mechanism and which do not, so that the overhead this recovery techniques involve is avoided where not needed. We believe that the information this tool provides can be used to find a way to automatically identify the parts of the client model that may benefit from the automatic error recovery and discard the ones that do not.

These tasks of analyzing, identifying and classifying the constraints managed with this tool are a first step. With this support, we can use this information to automatically modify the domain model and its constraints in a way that the resulting client can validate as many constraints as possible, minimizing communication with the server, and relieving the designer from finding out the required transformations that can be deduced automatically. Techniques for the automatic modification of constraints and domain models to achieve a more efficient integrity checking have already been studied, like the ones proposed by Cabot and Teniente [27], as well as techniques for adapting OCL constraints after the modification of UML models like Hassam et al. [26] proposals. We believe we can adapt some aspects of these techniques for our future needs regarding the automatic modification of constraints.

ACKNOWLEDGMENT

This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation: project TIN2011-25978 entitled Obtaining Adaptable, Robust and Efficient Software by including Structural Reflection to Statically Typed Programming Languages.

REFERENCES

[1] J. Duhl, “White paper: Rich internet applications,” Tech. report, IDC, 2003.

[2] J. Allaire, “Macromedia Flash MX—A next-generation rich client,” Macromedia white Pap., no. March, 2002.

[3] J. Garrett, “Ajax: A New Approach to Web Applications | Adaptive Path,” 2005. [Online]. Available from: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/ 2014.08.20

[4] A. Bozzon, S. Comai, P. Fraternali, and G. Carughi, “Conceptual modeling and code generation for rich internet applications.,” in Proceedings of the 6th international conference on Web engineering ICWE 06, 2006, vol. 1, p. 353.

[5] J. Preciado and M. Linaje, “Designing rich internet applications with web engineering methodologies,” Web Site Evol., pp. 23–30, 2007.

[6] A. Mesbah and A. Van Deursen, “An Architectural Style for Ajax,” in 2007 Working IEEEIFIP Conference on Software Architecture WICSA07, 2006, pp. 9–9.

[7] J. Louwsma, S. Zlatanova, R. Lammeren, and P. Oosterom, “Specifying and Implementing Constraints in GIS—with Examples from a Geo-Virtual Reality System,” Geoinformatica, vol. 10, no. 4, pp. 531–550, Jan. 2007.

[8] Z. L. Z. Liang and S. J. S. Jianling, “A field-oriented approach to web form validation for Database-Isolated Rule,” in 2009 IEEE International Conference on Systems Man and Cybernetics, 2009, no. October, pp. 4607–4612.

[9] “Object Management Group: UML 2.4.1 Superstructure Specification.” [Online]. Available from: http://www.omg.org/spec/UML/2.4.1/ 2014.08.20

[10] J. Warmer and A. Kleppe, The OCL, Second edition. Addison-Wesley, 2003.

[11] “Eclipse Modeling Framework Project - EMF.” [Online]. Available from: http://www.eclipse.org/modeling/emf/ 2014.08.20

[12] S. Hallé and R. Villemaire, “Browser-based enforcement of interface contracts in web applications with BeepBeep,” Comput. Aided Verif., pp. 648–653, 2009.

[13] P. Heidegger and P. Thiemann, “JSConTest: Contract-Driven Testing and Path Effect Inference for JavaScript.,” J. Object Technol., vol. 11, no. 1, p. 6:1, 2012.

[14] W. Z. W. Zhang, “2-Tier Cloud Architecture with maximized RIA and SimpleDB via minimized REST,” Comput. Eng. Technol. ICCET 2010 2nd Int. Conf., vol. 6, pp. V6–52–V6–56, 2010.

[15] A. Leff and J. Rayfield, “Programming model alternatives for disconnected business applications,” Internet Comput. IEEE, no. June, pp. 50–57, 2006.

[16] Z. L. Z. Liang and S. J. S. Jianling, “A field-oriented approach to web form validation for Database-Isolated Rule,” in 2009 IEEE International Conference on Systems Man and Cybernetics, 2009, no. October, pp. 4607–4612.

[17] K. Schmidt, R. Stühmer, and L. Stojanovic, “Gaining reactivity for rich internet applications by introducing client-side complex event processing and declarative rules,” in AAAI 2009 Spring Symposium: Intelligent Event Processing, 2009, pp. 67–72.

[18] “Drools - JBoss.” [Online]. Available from: http://drools.jboss.org/ 2014.08.20

[19] “IBM - ILOG,” Mar-2014. [Online]. Available from: http://www.ibm.com/software/info/ilog/ 2014.08.20

[20] "Apache Struts 2 Validation." [Online]. Available from: http://struts.apache.org/development/2.x/docs/validation.html 2014.08.20

[21] "jQuery Validation Plugin." [Online]. Available from: http://jqueryvalidation.org/ 2014.08.20

[22] "Simfatic Forms." [Online]. Available from: http://www.simfatic.com/ 2014.08.20

[23] E. Tilevich and Y. Smaragdakis, "J-orchestra: Automatic java application partitioning," in ECOOP '02 Proceedings of the 16th European Conference on Object-Oriented Programming, 2002, pp. 178–204.

[24] G. Hunt and M. Scott, "The Coign automatic distributed partitioning system," in OSDI '99 Proceedings of the third symposium on Operating systems design and implementation, 1999, no. February, pp. 187–200.

[25] F. Yang et al., "A unified platform for data driven web applications with automatic client-server partitioning," in Proceedings of the 16th international conference on World Wide Web - WWW '07, 2007, p. 341.

[26] K. Hassam, S. Sadou, V. Le Gloahec, and R. Fleurquin, "Assistance System for OCL Constraints Adaptation during Metamodel Evolution," in 2011 15th European Conference on Software Maintenance and Reengineering, 2011, pp. 151–160.

[27] J. Cabot and E. Teniente, "Incremental integrity checking of UML/OCL conceptual schemas," J. Syst. Softw., vol. 82, no. 9, pp. 1459–1478, Sep. 2009.

[28] M. Quintela-Pumares, D. Fernández-Lanvin, R. Izquierdo, and A.-M. Fernández-Álvarez, "Implementing automatic error recovery support for rich web clients," in WISE'10 Proceedings of the 11th international conference on Web information systems engineering, 2010, pp. 630–638.