MDA and Analysis of Web Applications

Behzad Bordbar and Kyriakos Anastasakis

School of Computer Science, University of Birmingham, Birmingham, B15 2TT, UK
{B.Bordbar,K.Anastasakis}@cs.bham.ac.uk

Abstract. Enterprise systems are mission critical. As a result, ensuring their correctness is of primary concern. This paper aids to the analysis of Web Applications, focusing on the aspects related to the interaction of business logic and Web browsers. The method adopted is based on the Model Driven Architecture. First, the Platform Independent Model of Web Applications is refined to create a new model called Abstract Description of Interaction (ADI). An ADI is a UML class diagram annotated with OCL statements to represent an abstraction of the interaction between the thin client and the business logic. Secondly, the ADI model is automatically transferred to an Alloy model and analysed using the Alloy Analyser.

1 Introduction

Over the past two decades Web Applications have become increasingly vital, affecting almost all aspects of our daily life such as banking, retail, information gathering, entertainment and learning. Such applications are mostly mission critical [1]. Hence, ensuring the correctness of the specification and implementation is a primary concern and has received considerable attention [2–4]. To analyse these systems, it is important to create a formal model. For example, [2] uses μ-calculus to represent the model, while [3] makes use of a variant of automata as the analyzable model. Stotts and Navon [4] present a model based on Petri nets. Our approach makes use of MDA [5–7] transformations to automatically create the analysable model. Unlike other approaches [2–4] we use a formalism [8], which is ideal [9] for the analysis of object oriented systems, such as web applications.

This paper aids to the analysis of Web Applications [10]; software applications, which are accessed via Web browsers. In particular, we are interested in identifying bugs such as the Amazon bug [11] and the Orbitz bug [12], which are created as a result of the interaction between browsers and the business logic. Figure 1 sketches our approach.

The MDA [5–7] emphasises the role of models by capturing high level abstraction of the system that is independent of any implementation platform, called Platform Independent Model (PIM). A PIM is then transformed to one or more Platform Specific Model (PSM) via an MDA tool. A PSM specifies the system in a particular implementation technology, platform and paradigm. There are already commercial [13] and non-commercial [14] MDA tools facilitating the

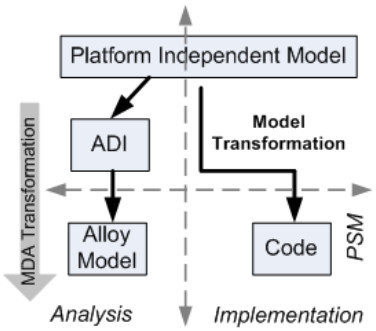

Fig. 1. Implementation and Analysis via MDA

implementation of a large part of the code on various choices of platform and programming languages. To analyse the model, the PIM has to be refined and abstracted to create a new PIM which we shall refer to as Abstract Description of Interaction (ADI). The ADI is a class diagram [15] with a set of OCL [16] constraints and pre and post condition expressions, that describes the interaction of the browser and the business logic in an abstract way. The ADI model can be translated to a model in Alloy [8] and analysed by the Alloy Analyser [17]. We have implemented the transformation from the UML to Alloy in a tool called UML2Alloy [18, 19].

The paper is organised as follows. Section 2 presents a brief introduction to MDA, Alloy and UML2Alloy. Section 3 sketches a method for the creation of the ADI. To demonstrate our approach, Sect. 4 analyses an example of an e-commerce system. Finally in Sect. 5 we sketch the related work and future direction, while Sect. 6 provides the conclusions of our work.

2 Preliminaries

Model Driven Architecture (MDA) [5–7] is a framework proposed by the Object Management Group [20]. Central to the MDA is the idea of model transformation, which maps models in a *source* language into a model expressed in a *destination* language. Models in the MDA are instances of *metamodels*. A metamodel is in effect a model that describes another model. The Meta Object Facility (MOF) [21] specifies the layered architecture that the MDA follows, where each model is an instance of its metamodel. As depicted by Fig. 2, an MDA transformation is defined from the source metamodel to the destination metamodel. Then every model, which is an instance of the corresponding metamodel, can be transformed to an instance of the destination metamodel. For example, to map a UML class diagram to Alloy, an MDA transformation that maps the metamodel of Class diagrams to the metamodel of the Alloy language is required. An MDA tool is a tool that implements a model transformation. In other words, it receives a description of the metamodels of the source and destination and a specifica-

tion of the model transformation rules and for every model that conforms to the source metamodel, it generates a corresponding model, which is an instance of the target metamodel.

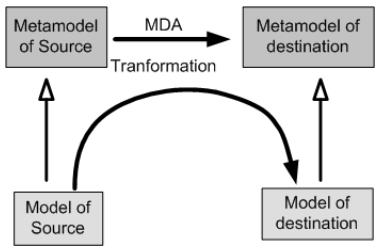

Fig. 2. Model transformation in the MDA

2.1 Alloy

Alloy [8] is a textual modelling language based on first order predicate logic [22]. An Alloy model is made of a number of *Signatures*, which describe the entities in the model. Signatures are similar to classes in a UML Class diagram. For example a Signature can define *Fields* which are like Attributes of classes in UML class diagrams.

There are also three major kinds of expressions in an Alloy model. A *Fact* is an expression that every instance of a model satisfies. *Predicates* and *Functions* are like functions in an object oriented programming language. They can be invoked from other parts of the model. Finally an *assertion* is a statement that the modeller wants to check for its validity. Alloy models are analysable and Alloy Analyzer [17], which is an implementation supporting the Alloy language, can present a *counterexample* if an assertion is violated.

Alloy tackles the state explosion problem [23] by introducing scoped analysis. A *scope* is the maximum number of stages the Alloy Analyzer probes to ensure the validity of an assertion or to find the existence of counterexamples. If the Alloy Analyzer fails to come up with a counterexample, the assertion may be valid. The bigger the scope is, the more confident the modeller is that his model is correct. Further details on the Alloy language the interested reader is referred to [8].

2.2 Analysis of UML Models via UML2Alloy

The UML is a family of languages that is prevailing in the modelling and specification of object oriented systems. The UML defines a number of diagrams [15], some of which depict the static structure of a system, while others the dynamic aspect. In this paper, we shall make use of UML class diagrams to model the

static structure of a system. We shall describe the behaviour of the system via OCL [16]. OCL is a textual language that adds formalism to UML diagrams. It can be used to define the behaviour of a model (with the use of *preconditions* and *postconditions*) or to express constraints (using *invariants*) on the elements of a UML model.

Based on the MDA, we have developed a CASE tool called *UML2Alloy* for automating the translation of UML models to the corresponding Alloy models. Figure 3 depicts the sequence of steps involved in the transformation. The starting point is to create a UML model of the system in a UML CASE tool such as ArgoUML [24]. Most UML tools, including ArgoUML, can export the UML model to an XMI [25] format. XMI, which stands for XML Metadata Interchange is an OMG standard used by most UML tools to store, import and export UML models. UML2Alloy implements the transformation and generates an Alloy model from the XMI file. The Alloy model of the system can then be analysed with the Alloy Analyser [17]. For further details on UML2Alloy, we refer the reader to [18, 19].

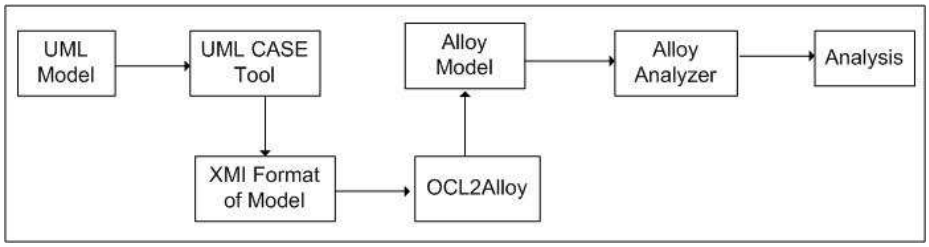

Fig. 3. Process of Analysis of UML models via UML2Alloy

In the following Section we shall present a method of verifying Enterprise Web Applications with the help of UML2Alloy.

3 Analysis of the Interaction Between a Browser and Business Logic

A Web Application is a software application that is accessible via a thin client (web browser). Web Applications often consist multiple tiers; the presentation tier (web server), application tier (business logic) and resource management (data) tier. Web Applications render web pages, comprising of different kinds information (e.g. text, images, forms) and are accessed via web browsers. Web pages can be *static* (i.e. residing on a web server) or *dynamic* (i.e. modifiable as a result of the execution of various scripts and components at the client or the server). As a result models of Web Applications are very large and complex, involving numerous components. Consequently, to conduct any realistic analysis, it is crucial to get rid of the unrelated information and create an abstract model

capturing the interaction between the browser and the business logic. To do so, we shall introduce a new class model called Abstract Description of Interaction (ADI). The following sketches the steps involved in the creation of the ADI.

1. *Browser* class: A model of the browser with its related functionality, such as navigating between various pages using the *back* and *forward* buttons. The browser class is an abstract model of the browser.
2. *Container* class: Models the generic functionality of the web pages comprising of data that can be dynamically altered from the user interface.
3. *BusinessLogic* class: An abstraction of the part of the business logic that relates to browser and its data content.
4. *Data* class: Describing the abstraction of the data that is exchanged between the server and the browser usually as fields of forms of web pages.

Figure 4 represents the interaction between the above classes inspired by the general form of Web Applications, in which:

- A *Browser* displays a *Container*.
- A *Container* can have a *previous Container*. To simplify we have not included a *next Container*.
- A *Browser* interacts with the *BusinessLogic*.
- A *Container* displays *Data* units
- A BusinessLogic deals with a number of *Data* units.

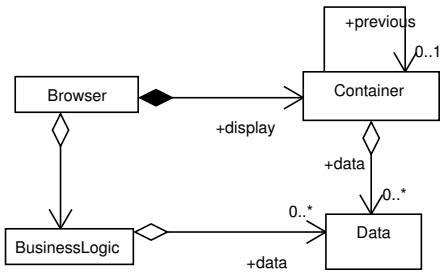

Fig. 4. A template for Abstract Description Interaction

The template of Fig. 4 and the steps sketched above can allow the modeller to probe the PIM and create the ADI. Platform Independent Models of Web Applications are large and complex. Hence, finding methods of a partial automation of the creation of the ADI is of paramount importance. Since the creation of the ADI involves the projection of the model and the deletion of unrelated model elements, we speculate that it might be possible to mark unrelated model elements on the PIM and refactor [26] the model to create the ADI, or a model near enough to the ADI. However that remains an area for future research. For now, to demonstrate examples of the creation of the ADI, we shall present a case study.

4 Analysis of an e-Commerce System: A Case Study

This case study is inspired by [27]. The class diagram of Fig. 5 represents a portion of the PIM of an internet bookstore system, extracted from [27, p. 120–123], which describes how the model of the system can be created following a process called ICONIX. In ICONIX information which is presented in a browser is stereotyped as *boundary*. The information that belongs to the business logic is stereotyped as *entity*.

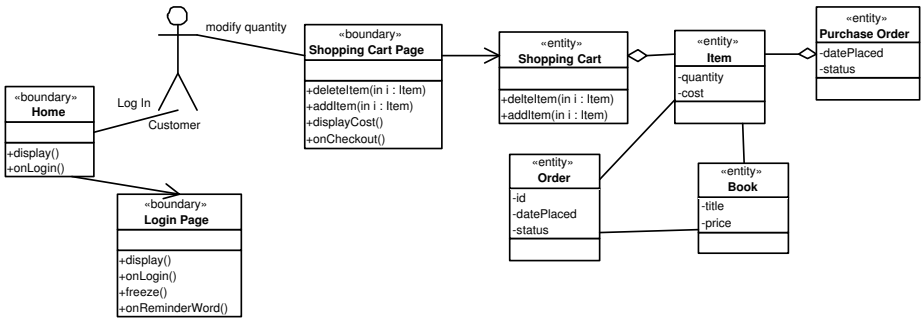

Fig. 5. UML model of an online bookstore

Our aim is to analyse the interaction of the user, through the web browser, with the business logic of the the online bookstore system modelled by Fig. 5. According to the method described in the previous Section, we have to get rid of the classes that are not related to our aim (i.e. they do not contain any functionality that affects the items in the shopping cart). It is obvious from the model that those classes are the *Home*, the *Login Page*, the *Book*, *Order* and *Purchase Order*. Of course depending on the business logic of the system, the *Purchase Order* might affect the quantity of the items in the shopping cart (i.e. the shopping cart might be emptied after the user has purchased an item), but for reasons of simplicity we are not going to consider this case.

Therefore we are now left with the *Shopping Cart Page*, the *Shopping Cart* and the *Item* classes. We now need to identify which of these classes are used for displaying information on the browser of the user and which for the business logic. However in our case study this is a trivial task as the classes that are used for displaying information to the user are stereotyped as *boundary* and the classes that are part of the business logic are stereotyped as *entity*. Therefore an abstraction of the specification of the functionality of the *Shopping Cart Page* will be used as the specification of the functionality of the *Container*. Similarly an abstraction of the specification of the functionality of the *Shopping Cart* will be used as the specification of the functionality of the *ShoppingCart*. The *Items* are the *Data* the browser exchanges with the business logic. However even from those classes we just need the functionality that changes the contents of the

shopping cart. Therefore we can safely get rid of some of the operations of those classes, such as the *displayCost()* and *onCheckout()* of the *Shopping Cart Page* class.

Following this reasoning, which is just a practical application of the method described in the previous Section, we end up with the model of Fig. 6.

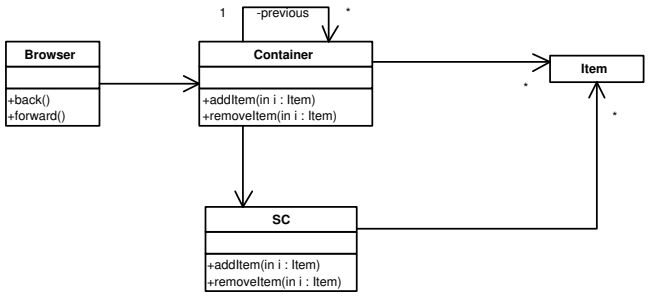

Fig. 6. Abstarct Description of Interaction: version 1

For practical reasons we had to refactor the model of Fig. 6. As OCL is a language used purely for the specification of systems, it does not allow the specification of an operation to "call" another operation that is not a query operation. A query operation is an operation that does not change the state of the system during its evaluation (e.g. an accessor to an attribute). [16, p. 5] This means that we are not allowed to "call" the *addItem* operation of the *ShoppingCart* from within the *addItem* opeartion of the *Container*, since the former changes the state of the system during its evaluation, by adding one item to the *ShoppingCart*. In order to overcome this constraint and for reasons of simplicity in the model, we have moved the specification of the functionality of the operations of the *Container* to the *Browser* class. We also consider a smooth communication channel between the browser and the web server. This enables us to also move the specification of the *addItem* and *removeItem* of the *Shopping Cart* to the *loadItem* and *removeItem* methods of the *Browser* respectively. This change to the ADI does not affect the model of the functionality of the system and the altered model that will be used for the analysis of the system is depicted in Fig. 7.

The *Browser* is related to a *Container*, which represents the web page the *Browser* displays. We are only interested in the items the user can buy from the web page. In the model those items are depicted with the *cHasItems* relation. Those are the items in the shopping cart the user sees on his/her web browser. The *Browser* is also related with the *SC*, which is the shopping cart. The *SC* represents the information held on the server regarding the items the user has added in the shopping cart. The *SC*, like the *Container*, is related to zero or more *Items*.

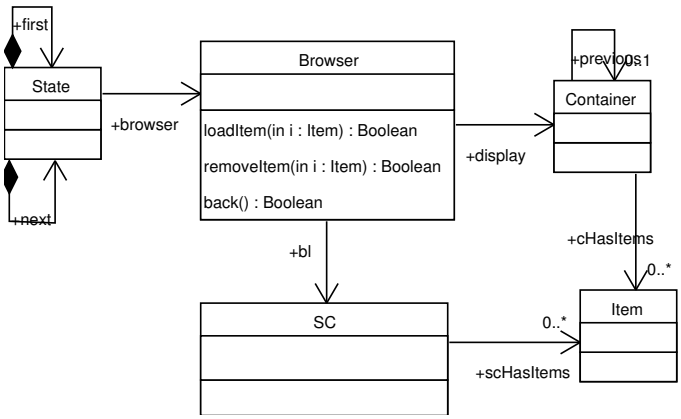

Fig. 7. Abstarct Description of Interaction

The *Browser* is a class that represents the web browser the user uses to access the web site. The *Browser* class has three operations. The *loadItem* contains the specification that describes the functionality of the system when the user adds an item, i to the shopping cart. More specifically the *loadItem* operation adds the item to the collection of items the *Container* displays to the user as the shopping cart and also adds it to the collection of items of the shopping cart that the server holds for that session. It also causes a new *Container* to be displayed and the old *Container* is set to be the *previous* of the currently displayed *Container*. In principle this action adds the previously displayed web page to the web browser history. The *removeItem* similarly contains the specification for the functionality of the system when the user removes an item i from the shopping cart. The invocation of the *removeItem* opration removes the item from the shopping cart the browser displays, adds the page being displayed to the browser history and invokes a communication with the web server shopping cart informing it that the item has been removed. The *back()* operation specifies the functionality of the system, when the back button of the *Browser* is pressed. In particular it causes the *previous Container* to be displayed, if any exists. Figure 8 depicts the OCL specification of the *back* operation. It is important to note that the use of the back button in any web browser invokes the previous request sent to the server and also uses cached information locally.

```
context Browser::back ()
pre: self.display.previous -> size > 0
post: self.display = self.display@pre.previous
```

Fig. 8. Behaviour of “back” button via OCL

State is a reserved class in UML2Alloy. Like any other class, UML2Alloy transforms the UML class *State* to an Alloy signature with the same name. However if the *State* class exists in a model UML2Alloy makes use of the polymorphic ordering module distributed with the Alloy Analyzer, which provides support for ordered sets. This is the standard way to achieve process modelling in Alloy [28].

There is also an OCL expression related to the *State* class that depicts the initial state of the system. In that state the shopping cart of the *Container* of the *Browser* does not display any items and the information about the shopping cart that is held on the server does not have any items as well.

The behaviour of the model is depicted by the OCL statement of Fig. 9. The statement defines that for all ($\forall$) States s there exists ($\exists$) at least one item i in the model so that the next State *s'* can be produced from the previous if either the user adds an item to the shopping cart (the *loadItem* operation) or removes an item (the *removeItem* operation) , or the "back" button of the browser is pressed (the *back()* opeartion). In that case as explained before the interaction with the server does not cause the information about the shopping cart that is held on the server to change as depicted by the statement $s.next.browser.bl = s.browser.bl$.

```
context State
State.allInstances -> forAll(s:State |
(s.browser.display.cHasItems -> exists(
i:Item |(
(s.next.browser = s.browser.loadItem(i))
or
(s . next . browser = s . browser . removeItem( i ) )
or
((s.next.browser = s.browser.back())
and (s.next.browser.bl = s.browser.bl ))))))
```

Fig. 9. A portion if the behaviour of the model in OCL

4.1 Produced Alloy Model

Figure 10 depicts the corresponding Alloy model for the OCL statement of Fig. 9. Naturally, *forAll* and *exists* of OCL are mapped to *all* and *some*. Moreover, the subsequent state of s, i.e. $s.next$, is mapped into the state $s' : ord/next(s)$. However, automated transformation from OCL to Alloy is far from trivial. In OCL, the invocation of an operation is through navigate to the class that owns the operation, and then "calling" the operation. In contrast, Alloy predicates and functions are visible to the whole model. Therefore, during the transformation all OCL statements that invoke operations from another class had to be tailored

so as to translate only the part that calls the operation. It can also be noted that more parameters have been added to the *loadItem*, *removeItem* and *back()* operations. This is the usual pattern of specifying pre and post conditions in the Alloy language. For further details regarding the transformation from OCL to Alloy, we refer the reader to [19].

```
all s: State ,s':ord/next(s)| some i:Item |
(loadItem(s.browser,s'.browser,i)) ||
(removeItem(s.browser,s'.browser,i)) || (back(s.browser,s'.browser)
  &&
(s'.browser.bl=s.browser . bl) )
```

Fig. 10. Alloy model of the behaviour

4.2 Results of the Analysis

A major requirement of the model depicted by Fig. 7 is to guarantee the integrity of the system by ensuring that the contents of the shopping cart and the list of items on the browser are identical. This can be expressed by the Alloy *assertion* of Fig. 11. Using Alloy Analyzer we can see that the assertion fails. In fact, setting

```
all s: State | s.browser.display.cHasItems = s.browser.bl.scHasItems
```

Fig. 11. Alloy assertion for checking shopping cart against the displayed items

the Alloy *scope* to three, results in a counterexample, which means, there is at least one instance in three evolutions of the system where the items listed in the browser window differs from the items in the shopping cart.

Using Alloy Analyzer the modeller can see not only the instance that violates the assertion, but also all of the traces in the evolution of the system that lead to the violation of the assertion. That way it is easier to locate the inconsistency in the model. The steps to reproduce the bug according to our analysis are the following:

Step 1 The shopping cart of the user is empty and the user browses the web site.

Step 2 The user adds an item *Item1* to the shopping cart.

Step 3 The user decides that he does not want to buy *Item1* after all, but instead of deleting it from the shopping cart he presses the "back" button to return to the previous shopping cart which is empty.

Comparing the result of our analysis with the motivating observation of the paper [11], we can notice that the bug can occur following a different trace from the one described in [11], but again it involves the use of the "back" button. To remove such bugs, [11] presents a set of solutions. Analysing such solutions is outside of the scope of the paper. However, our method can equally be used to conduct such analysis.

5 Related and Future Work

Because of the state explosion problem [23] and undecidability issues, it is not possible to analyse large systems, such as Enterprise Web Applications. A method to overcome this problem is to either partially analyse the system, focusing on different aspects of the system every time or to abstract the model that represents it. Using this approach we have decided to focus on the analysis of the ADI, an abstract view of the part of model of the system that depicts the interaction between the user and the business logic. Such analysis increases our confidence of the correct functionality of the system. We are currently working on methods of creating the ADI via semi-automated methods, using refactoring methods [26].

A highly promising direction is to adopt form-based approach [29]. *Form-charts* as a formalism developed for the modelling of form-oriented applications is ideal for capturing the interaction of the user interface with the business logic. As a result, it might be possible to use MDA transformations to create a suitable formchart from which the ADI can be inferred. However, this remains a subject for future work.

6 Conclusions

This paper aids to the analysis of Enterprise Web Applications. The method adopted draws on the Model Driven Architecture. The Platform Independent Model of systems can be used to create the ADI model, an abstraction of the interaction between the browser and the business logic of the system. In order to analyse the ADI, we apply a further MDA transformation and create a corresponding Alloy model. The paper demonstrates that analysing the ADI, it is possible to identify a group of bugs, such as the Amazon bug [11]. Finally the approach presented in the paper is explained via an example of an e-commerce system.

References

1. Fowler, K.: Mission-critical and safety-critical development. IEEE journal of Instrumentation & Measurement Magazine **7** (2004) 52– 59
2. de Alfaro, L.: Model checking the world wide web. In: Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings. Volume 2102 of Lecture Notes in Computer Science. (2001) 337–349

3. Haydar, M., Petrenko, A., Sahraoui, H.A.: Formal verification of web applications modeled by communicating automata. In: Proceeding of Formal Techniques for Networked and Distributed Systems - FORTE 2004, 24th IFIP WG 6.1 International Conference, Madrid Spain, September 27-30, 2004. Volume 3235 of Lecture Notes in Computer Science. (2004) 115–132
4. Stotts, D., Navon, J.: Model checking cobweb protocols for verification of html frames behavior. In: Model checking cobweb protocols for verification of HTML frames behavior. (2002) 182 –190
5. MDA: (Model Driven Architecture website: http://www.omg.org/mda)
6. Frankel, D.S.: Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley Publishing, Indianapolis, Indiana (2003)
7. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture - Practice and Promise. The Addison-Wesley Object Technology Series. Addison-Wesley (2003)
8. Jackson, D.: Alloy 3.0 Reference Manual (May 2004) Software Design Group, MIT Lab for Computer Science, http://alloy.mit.edu/beta/reference-manual.pdf.
9. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (TOSEM) **11** (2002) 256–290
10. W3C: (W3C website: http://www.w3.org/)
11. Baresi, L., Denaro, G., Mainetti, L., Paolini, P.: Assertions to better specify the amazon bug. In: SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering, New York, NY, USA, ACM Press (2002) 585–592
12. Licata, D.R., Krishnamurthi, S.: Verifying interactive web progams. In: 19th IEEE International Conference on Automated Software Engineering (ASE 2004), Linz, Austria, IEEE Press (2004) 164–173
13. Interactive Objects: (Archstyler website: http://www.interactive-objects.com/)
14. AndroMDA: (AndorMDA website: http://www.andromda.org/)
15. Object Management Group: (Unified Modeling Language v 2.0 Superstructure Final Adopted Specification) Document id: ptc/03-08-02. http://www.omg.org/docs/ptc/03-08-02.pdf.
16. Object Management Group: (UML 2.0 OCL Final Adopted Specification) Document id: ptc/03-10-14. http://www.omg.org/cgi-bin/doc?ptc/2003-10-14.
17. Alloy Analyzer: (Alloy Analyzer website: http://alloy.mit.edu/)
18. UML2Alloy: (Uml2alloy website: http://www.cs.bham.ac.uk/∼bxb/UML2Alloy.php)
19. Bordbar, B., Anastasakis, K.: UML2Alloy: A tool for lightweight modelling of Discrete Event Systems. In Guimarães, N., Isaías, P., eds.: IADIS International Conference in Applied Computing 2005. Volume 1., Algarve, Portugal, IADIS Press (2005) 209–216
20. Object Managemenet Group (OMG): (OMG website: http://www.omg.org)
21. Object Management Group: (Meta Object Facility (MOF) 2.0 Core Specification) Document Id: ptc/03-10-04. http://www.omg.org/cgi-bin/apps/do_doc?ptc/03-10-04.pdf [cited April 2005].
22. Jackson, D.: Automating first-order relational logic. In: SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering, New York, NY, USA, ACM Press (2000) 130–139
23. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets. Volume 1492 of LNCS., London, UK, Springer-Verlag (1998) 429–528

24. http://argouml.tigris.org/: (ArgoUML)
25. Object Management Group: UML 2.0 Diagram Interchange Final Adopted Specification (2003) Document Id:ptc/03-09-01. http://www.omg.org.
26. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: improving the design of existing code. Addison-Wesley, Boston, MA, USA (1999)
27. Rosenberg, D., Scott, K.: Applying Use Case Driven Object Modeling with UML: An annoted E-Commerce Example. Addison-Wesley Object Technology Series. Addison-Wesley (2001)
28. Wallace, C.: Using Alloy in process modelling. Information and Software Technology **45** (2003) 1031–1043
29. Draheim, D., Weber, G.: Form-Oriented Analysis: A New Methodology to Model Form-Based Applications. Springer-Verlag, Berlin, Germany (2005)