UML ANALYSIS USING STATE DIAGRAMS

Mohammad N. Alanazi, Jason A. Belt, Dr. David A. Gustafson
Department of Computing and Information Sciences
College of Engineering
Kansas State University
Manhattan, KS, U.S.A
{alanazi, belt, dag}@ksu.edu

Abstract — This article demonstrates a new approach to analyzing UML designs using state and sequence diagrams. From multiple state diagrams, a super-state diagram which includes the cross-product of the selected states is built along with a transition matrix of possible transitions. The closure of the transition matrix is used to identify unreachable states and impossible transitions. Additionally, the closure is also used to evaluate consistency between the state diagrams and the sequence diagrams. Missing and impossible sequences can be identified. A prototype tool has been built to calculate the closure of the transition matrix and to compare the results with the sequence diagrams.

1. Introduction

Finding errors in software designs before they are implemented is very important. Most researchers in software engineering have found that the earlier an error is found, the easier it is to correct. If we can improve the analysis capabilities for software designs, we may have a significant impact on removing faults earlier in the software development process.

The typical software design is specified using UML diagrams [7] including class diagrams, use case and sequence diagrams, and state diagrams. Although class diagrams and use case/sequence diagrams are the most common diagrams in a software design, the state diagram provides an excellent notation for specifying the behavior of objects and how methods affect the objects.

Sequence diagrams are interclass (interobject or interagent) and they detail how objects in the model interact via method calls. A sequence diagram can be viewed as a partial collaborative view of a set of objects. Transitions in a sequence diagram occur as the result of method calls between objects. Methods of different objects are often paired together as the result of one object's method calling the method of another object.

A state diagram is intraclass (intraobject or intraagent) and it describes the states one object in the model can be in and the transitions which cause that object to change state. A state diagram provides the component view of an object. To understand how the states of one object interact with the states of another object, a different approach has to be used.

The UML specification does not enforce any consistency requirements between the information contained in the sequence and state diagrams. While this does allow for greater flexibility in how UML can be used, it can lead to inconsistent views of the system being modeled. The problem of relating state diagrams with sequence diagrams has recently been the focus of research in the software engineering community [e.g. 1]. However, the work has usually involved just one state diagram and one or more sequence diagrams.

Our approach to consistency analysis combines the state information of multiple state diagrams into a composite super-state diagram. This super-state diagram details all of the possible composite states the objects can be in as well as the transition pairs which lead from one composite state to another. In this way the super-state diagram provides the complete collaborative view of a set of objects in the model. A given sequence diagram then should be a valid subsequence of the set of sequences that are possible in a super- state diagram.

2. Transition Matrix

The basis of our analysis techniques is a transition matrix that details the possible global states of the system based on a vector of states of individual instances of classes and the possible transitions between the states in the global state vector. Consider a program that has class X and class Y. Let class X have an initial state A and two other states, B and C, while class Y has an initial state D and a second state E. The state diagrams will depict how instances of X and Y can transition between those states. Let class X call class Y whenever class X transitions between state A and B. Table 1 shows possible transitions in the super-state diagram that is the cross-product of all states with one instance of X and one instance of Y.

Table 1 - Super-state transition matrix T1.

T_1	AD	BD	CD	AE	BE	CE
AD	0	0	0	0	1	0
BD	1	0	1	0	0	0
CD	0	1	0	0	0	0
AE	0	1	0	0	0	0
BE	0	0	0	1	0	1
CE	0	0	0	0	1	0

An entry in a cell in T_1 (Table 1) shows that in one step, the system can transition from the state of the row to the state of the column. Taking the product of T_1 by itself gives a matrix that contains the transitions possible with two steps. The closure of T_1 is the sum of products, $T_1 + T_1*T_1 + T_1*T_1*T_1 + \ldots$.The closure shows all possible transitions in any number of steps. Although the closure is represented as an infinite sum, it can be calculated in at most the number of products equal to the rank of the initial matrix. In most cases, it is even smaller than that number.

3. Error Discovery

The closure of the transition matrix for a cross-product of a number of instances will be the basis for our analysis. The closure of the transition matrix by itself can show four kinds of errors:

1. existence of a bad combination of states
2. unreachability of a good combination of states
3. existence of a bad transition
4. unreachability of an important transition

The output from the closure of the transition matrix can also be compared with the class diagram and the sequence diagrams to look for consistency. Since all possible transitions are generated by the closure, every sequence that appears in a sequence diagram should be in the closure. Also, for completeness, every sequence in the closure should be represented by a sequence diagram. Although this is not absolute, it would be a good check for completeness of the sequence diagrams to check those closure matrix sequences to see if they warrant inclusion in a sequence diagram.

4. Library Example

This example describes the interaction between a patron of a library and the copies of books the library holds. In order to simplify the model the library holds only one copy of each book. Figure 1 is the class diagram for this model and Figure 2 and Figure 3 are the state diagrams for the patron and book objects. Note that the transitions in the state diagrams are numbered for ease of reference. This example along with the sequences diagrams was created by a team of students trying to create a correct model of a simple library system.

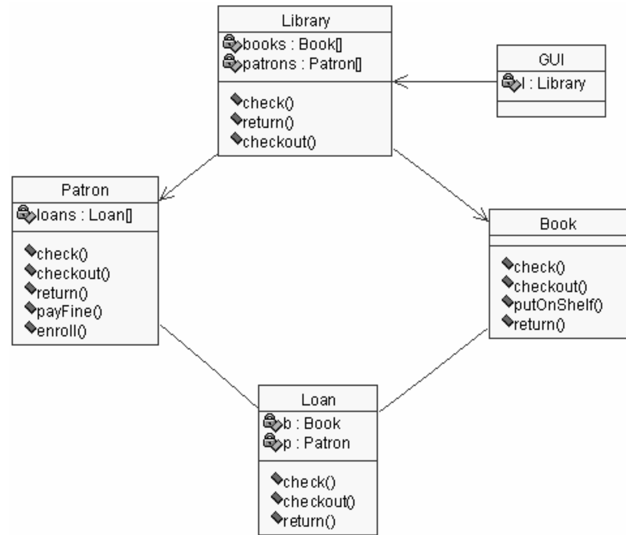


Figure 1 - Class Diagram for the library example

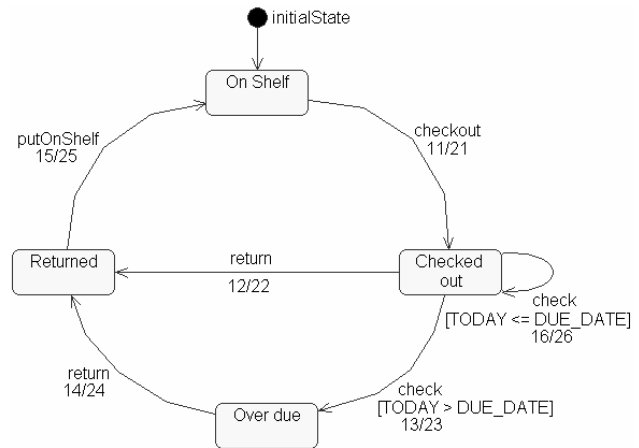


Figure 2 – State Diagram for Book

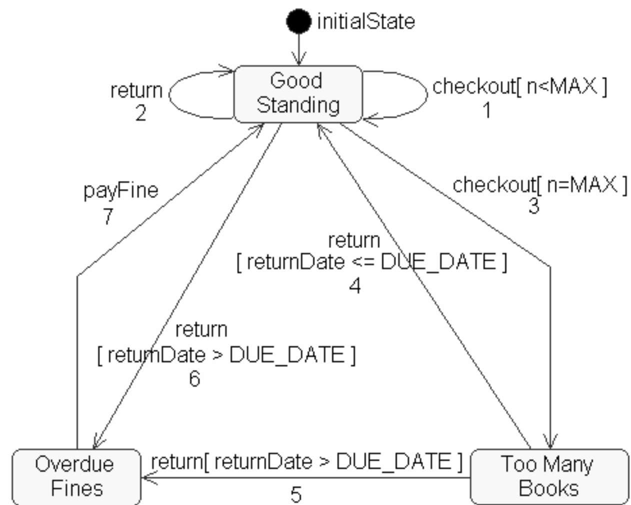


Figure 3 – State Diagram for Patron

The patron object can be in one of three states; *Good Standing*, *Too Many Books*, and *Overdue Fines*. We will call these states *G*, *T*, and *F* respectively for the rest of this paper. A patron starts in *G* until the number of books the patron has checked out is equal to *MAX* or the patron returns an overdue book. In the former, the patron will transition to state *T* where they will remain until they return a book. In the latter, the patron will transition to *F* where they will not be able to do anything until they pay the fine that is owed. Note that this does not allow the patron to return additional books if they currently owe a fine (a subtle and unintended difference from most actual library systems).

A book object has four states; *On Shelf*, *Checked Out*, *Overdue*, and *Returned*. We will call these states *O*, *C*, *D*, and *R* respectively for rest of this paper. When a book is checked out it transitions from *O* to *C*.

The two transitions from *C* labeled *check* represent the library determining if the book is overdue. If the book is overdue it will transition to *D*. Otherwise it will transition to *R* where it will remain until the library places it back on the shelf.

For our analysis we will assume the library has only one patron and two books. We now pair the transitions from the patron and book objects that can occur together. An 'X' indicates we are not concerned about the state of the object.

GOX → *GCX*	patron checks out a book
GXO → *GXC*	
GCX → *GRX*	patron returns a book on time
GXC → *GXR*	
GDX → *FRX*	patron returns an overdue book
GXD → *FXR*	
TCX → *GRX*	patron with *MAX* books returns a book
TXC → *GXR*	
TDX → *FRX*	patron with *MAX* books returns an overdue book
TXD → *FXR*	
GCC → *TCC*	patron attempts to check out *MAX* + 1 books

The following transitions can occur independent of the states of the other objects

F → *G*	patron pays fine
C → *D*	book becomes overdue
C → *C*	book remains checked out
R → *O*	book is re-shelved

The initial transition matrix A_1 has column and row headings with triples representing the states of the three objects. For this model there are $3 \cdot 4 \cdot 4 = 48$ combinations of the three objects. Table 2 shows a portion of the initial transition matrix A_1.

Table 2 - Portion of A_1

	GOO	*GOC*	*GOD*	*GOR*	*GCO*
GOO		1,21			1,11
GOC		26	23	2,22	
GOD					
GOR	25				
GCO					16

The row headings are the initial states and the column headings are the final states. The numbers in the table arise from Figure 2 and Figure 3. For the purpose of clarification we have assigned unique numeric identifiers to the transitions for each instance of an object in our system. The book object has two numeric identifiers for each transition since we have two instances of that object. For example, *GOO* → *GOC* represents a patron in good standing checking out the second book. The 1 indicates the patron took the transition labeled *checkout* [$n < MAX$] and the 21 indicates the second book took the transition labeled *checkout*. If there is an entry for a cell in the matrix then the transition is valid.

A_2 is defined as $A_1 \cdot A_1$ which identifies all the states we can reach in two steps. Table 3 shows a portion of A_2.

Table 3 - Portion of A_2

	GOO	*GOC*	*GOD*
GOO		(1,21)(26)	(1,21)(23)
GOC	(2,22)(25)	(26)(26)	(26)(23)
GOD			
GOR		(25)(1,21)	
GCO	(2,12)(15)		

From Table 3 we can observe that it is possible to go from *GOC* to *GOO* by first returning the second book and then shelving it.

The closure of *A* (i.e. A^*) for this model occurs after 5 multiplications of A_1. Since A^* will give all of the possible sequences through the combined state diagrams, we would expect the cells for the unreachable states to be empty.

For this model the unreachable states include two sets. The first set includes the states where the patron is in *T* and one of the two books is in *O* or *R*. Clearly the patron can not have *MAX* books checked out if one of the books is not checked out. The other set of unreachable states occurs when the patron is in *F* and one book is in *C* and the other is in *D* or both books are in *C* or *D*. In order for the patron to be in *F* one of the two books would have had to have been returned. An analysis of A^* for this model shows that the columns for these unreachable states are empty.

Some of the faults in the design of the library example can be discovered by simply analyzing the transition

matrix. One such fault was a missing transition. From *FRC* and *FCR* there is not valid single-step transition to *FRR*. This means that if one book is returned late, the patron goes to *F* status and can not return the other book until they pay the fine.

The remainder of the analysis of the library model will be done using the tool we have developed which implements our consistency checking approach.

5. UML Design Analysis Tool

The UML Design Analysis Tool is a tool to check consistency between diagram sets within a single Rational Rose Project and a Transition Set that is user defined. These XMI files are created from Rational Rose via XMI export. The file contains an XMI representation of all models created within the project set. The Transition Set is a text based file created by the user and is documented in a later section. These files are parsed into an appropriate data structure for validation based comparisons. A set of results is generated in the form of error messages and are displayed in text upon program completion.

The Transition Matrix is constructed using matrix arithmetic on the string values of transitions to populate the transitive closure. Multiplication is a logical AND on the string contents and addition is a logical OR.

Although many of the individual transitions are obvious, we have not yet automated the identification of these transitions. Instead, the user supplies the Transition Set file which is used to complete transition matrix. Figure 4 shows the Transition Set file for the library example:

```
//  P:Patron,b1:Book1,b2:Book2 > trans > P:Patron,b1:Book1,b2:Book2
//  and b3 representing a book more than the patron is allowed to check out
x, Returned, x > b1.putonShelf > x, On Shelf, x
x, x, Returned > b2.putonShelf > x, x, On Shelf
x, Checked Out, x > P.check,b1.check > x, Over due, x
x, x, Checked Out > P.check,b2.check > x, x, Over due
x, Checked Out, x > P.check,b1.check > x, Checked Out, x
x, x, Checked Out > P.check,b2.check > x, x, Checked Out
Good Standing, Checked Out, Checked Out > P.checkout,b3.checkout > Too Many Books,
     Checked Out, Checked Out
Good Standing, On Shelf, x > P.checkout,b1.checkout > Good Standing, Checked Out, x
Good Standing, x, On Shelf > P.checkout,b2.checkout > Good Standing, x, Checked Out
Good Standing, Checked Out, x > P.return,b1.return > Good Standing, Returned, x
Good Standing, x, Checked Out > P.return,b2.return > Good Standing, x, Returned
Good Standing, Over due, x > P.return,b1.return > Overdue Fines, Returned, x
Good Standing, x, Over due > P.return,b2.return > Overdue Fines, x, Returned
Too Many Books, Checked Out, x > P.return,b1.return > Good Standing, Returned, x
Too Many Books, x, Checked Out > P.return,b2.return > Good Standing, x, Returned
Too Many Books, Over due, x > P.return,b1.return > Overdue Fines, Returned, x
Too Many Books, x, Over due > P.return,b2.return > Overdue Fines, x, Returned
Overdue Fines, x, x > P.payFine > Good Standing, x, x
```

Figure 4 – Transition Set file

For the library example, an entry in the Transition Set file has the format:

Initial state, Initial state, Initial state > Transition(s) > Final state, Final state, Final state

Each state triplet represents a composite state of the three objects. By taking the transitions listed, the model will transition from the initial composite state to the final composite state.

If a user has no preference of the state of a particular object in a composite state, then an x can be used to denote "don't care." For example, to show the effect of calling *putOnShelf* on the first book, which has previously been returned, the user input line would read as:

x, Returned, x > putOnShelf > x, On Shelf, x

Here, the states of the patron and the second book will remain the same, but the first book will go from *R* to *O*.

6. Results From the Library Example

We now present the output we obtained after running the tool on several sequence diagrams developed for the library example.

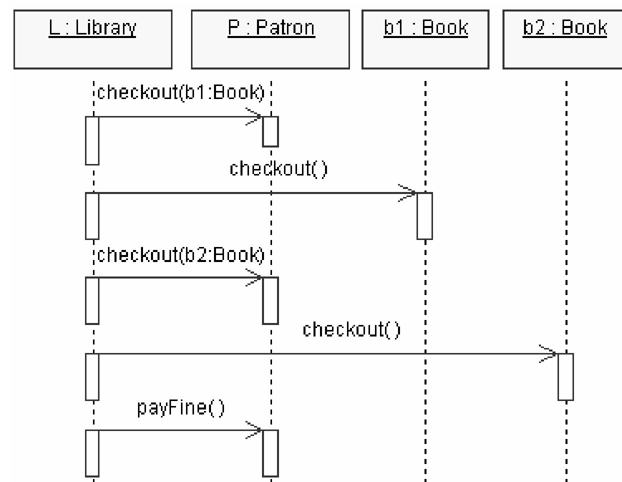


Figure 5 - Sequence for returning overdue book

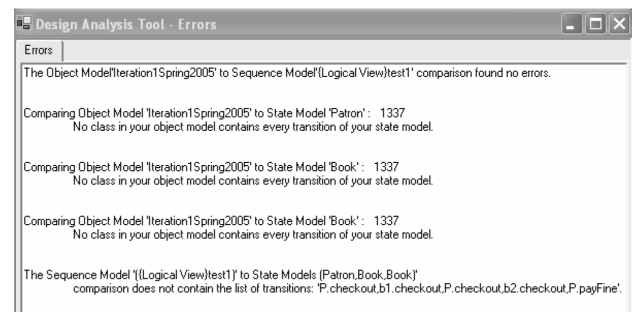


Figure 6 - Tool output for Figure 5 sequence diagram

The first sequence diagram (Figure 5) is checking out two books and returning an overdue book. The tool's output (Figure 6) shows that comparing the class diagram with the sequence diagram found that there was no association between patron and book. Additionally, it found that the sequence in the figure 5 was not legal since a check action on the book must occur before a book becomes overdue.

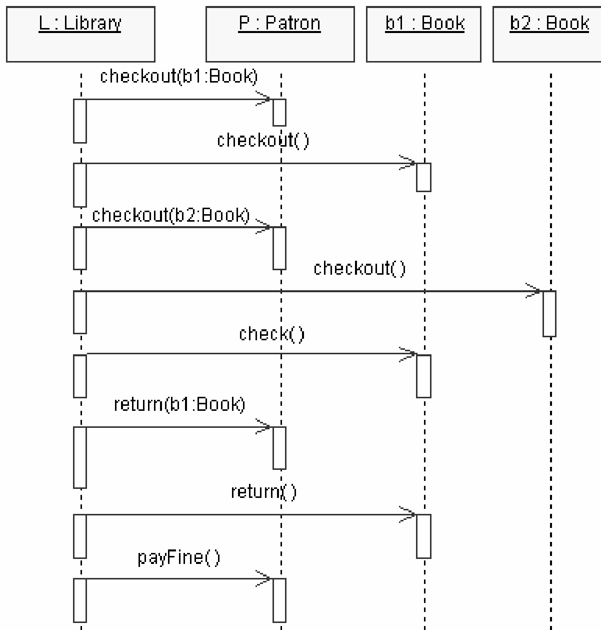


Figure 7 - A corrected sequence for overdue book

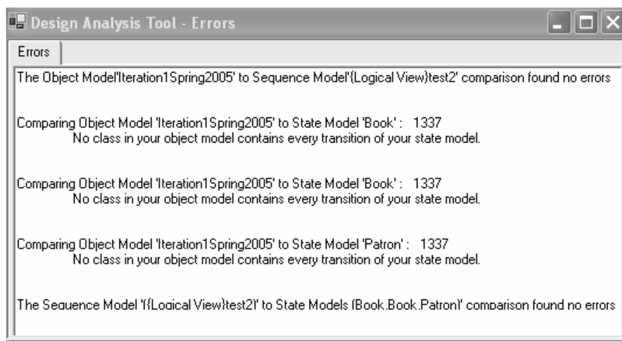
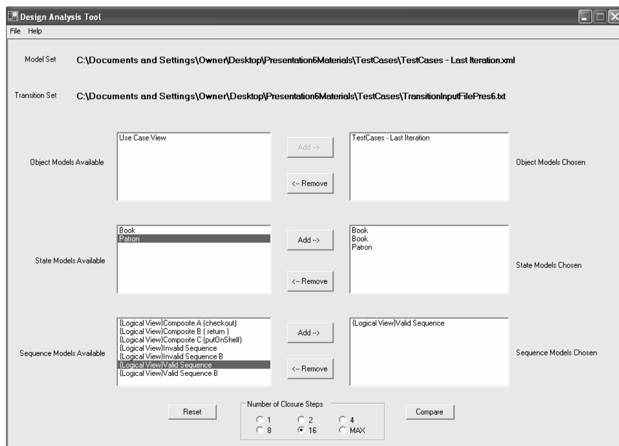


Figure 8 - Tool output for Figure 7 sequence diagram

Figure 8 shows that the tool correctly identified the sequences in Figure 7 as a correct set of sequences.

7. Screen Snapshots



8. Related Work

There are several different approaches that have been proposed to perform consistency checking between UML diagrams.

Boris Litvak et al. [6] presents an algorithmic approach to consistency checking between UML Sequence and State diagrams. They created the BVUML (Behavioral Validator of UML) tool which automates the behavioral validation process. Their approach associates states with the Sequence diagrams object lifeline so a single run of the tool validates consistency for only one object. Therefore the tool must be run multiple times in order to check the consistency of an entire sequence diagram.

Orest Pilskalns et al. [8] presents an approach that combines structural and behavioral UML representations in order to derive and execute test cases to validate a UML model. They develop a method for encapsulating the behavioral aspects (i.e. message paths between objects) that exists in sequence diagrams into a directed acyclic graph. The objects in the graph are then associated with class attribute/parameter values which are used to generate and execute test cases. Their approach would require OCL object constraints to be written.

Krishnan [5] describes a framework in which UML diagrams can be formalized to perform consistency checking. UML diagrams are translated into specifications of the theorem proving tool PVS (Prototype Verification System). The PVS is a language that allows for the introduction of abstract data types, functions etc. To check for consistency between sequence and class diagrams, the class diagrams must first be annotated with OCL constraints. The PVS will check if the sequence of states described in the sequence diagram can be obtained from the class diagrams. Custom traces (i.e. sequence of states) can also be supplied by the user to check if other properties hold.

Soon-Kyeong Kim and David Carrington [4] describes how consistency checking between different UML models can be accomplished by using a formal object-oriented metamodeling approach. They formally define the abstract syntax and semantics of the UML model using Object-Z as a metalanguage. They then define consistency constraints that logically exist between semantically equivalent elements in the metamodel but are not defined in the current UML metamodel structure. Once the consistency constraints have been defined for each of the UML model elements, consistency checking between different model elements can be achieved by verifying that the combined models preserve all of the consistency constraints for the individual model elements.

Alexander Egyed [3] presents a transformation-based approach to consistency checking. They define a set of model transformation rules to enable the conversion of one UML diagram into another. They also define a set of comparison rules to compare the transformed diagram with an existing one of the same type. For example, to check for inconsistencies between a sequence diagram and a class diagram, they first transform the sequence diagram into an interpreted class diagram. The interpreted class diagram is then compared with the existing class diagram.

Yves Dumond et al. [2] shows that it is possible to integrate semi-formal and formal methods for the dynamic behavior of the UML models. The objective is to favour the integration of formal techniques in the actual practice of software engineering. They introduce an approach to formalize sequence diagrams and verify coherence with the statechart diagrams. The approach translates the UML sequence diagrams into the pi-calculus, by preserving the object paradigms. The consistency between sequence diagrams and statechart diagrams can be checked by verifying that the messages in the sequence diagrams trigger states in statechart diagrams.

9. Conclusions and Future Work

In this paper we have shown how consistency checking between state and sequence diagrams can be accomplished through the use of a transition matrix for a vector of states representing the state of more than just one instance object. The closure of this transition matrix can be used to identify reachable and unreachable states, identify which transitions are valid, to validate class diagrams, and to validate sequence diagrams.

We also introduced the UML Design Analysis Tool which implements our consistency checking methodology. We have shown how our methodology accurately identified the sequence diagrams which were inconsistent with the state diagrams.

Further work needs to be done to show that these techniques can be used on more complex State Diagrams such as those that include concurrency and nested states. We are also working on automating the process of identifying the transition pairs.

10. References

[1] Bontemps, Y.; Heymans, P.; Schobbens, P.-Y. "From Live Sequence Charts to State Machines and Back: A Guided Tour". IEEE Transactions on Software Engineering, 31(12): 999--1014, Dec. 2005.

[2] Yves Dumond, Didier Girardet , Flavio Oquendo. "A relationship between sequence and statechart diagrams". Dynamic Behaviour in UML Models: Semantic Questions, UML 2000 Workshop.

[3] Alexander Egyed. "Scalable Consistency Checking between Diagrams – The ViewIntegra Approach". Proceedings of the 16th Annual International Conference on Automated Software Engineering, pages 387--390, 2001.

[4] Soon-Kyeong Kim and David Carrington. "A Formal Object-Oriented Approach to defining Consistency Constraints for UML Models". Proceedings of the 2004 Australian Software Engineering Conference, pages 87--94, 2004.

[5] Padmanabhan Krishnan. "Consistency Checks for UML". Proceedings of the Seventh Asia-Pacific Software Engineering Conference, pages 162--169, 2000.

[6] Boris Litvak, Shmuel Tyszberowics, and AmiramYehudai. "Behavioral Consistency Validation of UML Diagrams". Proceedings of the First International Conference on Software Engineering and Formal Methods, pages 118--125, 2003.

[7] OMG Unified Modeling Language Specification, Version 1.5, Object Management Group, 2003, http://www.uml.org

[8] Orest Pilskalns, Anneliese Andrews, Sudipto Ghosh, and Robert France. "Rigorous Testing by Merging Structural and Behavioral UML Representations". UML 2003 - The Unified Modeling Language: Modeling Languages and Applications, 2863: 234--248, 2003.