World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:2, No:2, 2008

Inconsistency Discovery in Multiple State Diagrams

Mohammad N. Alanazi, and David A. Gustafson

Abstract**—**In this article, we introduce a new approach for analyzing UML designs to detect the inconsistencies between multiple state diagrams and sequence diagrams. The Super State Analysis (*SSA*) identifies the inconsistencies in super states, single step transitions, and sequences. Because *SSA* considers multiple UML state diagrams, it discovers inconsistencies that cannot be discovered when considering only a single UML state diagram. We have introduced a transition set that captures relationship information that is not specifiable in UML diagrams. The *SSA* model uses the transition set to link transitions of multiple state diagrams together. The analysis generates three different sets automatically. These sets are compared to the provided sets to detect the inconsistencies. *SSA* identifies five types of inconsistencies: impossible super states, unreachable super states, illegal transitions, missing transitions, and illegal sequences.

Keywords**—**Modeling Languages, Object-Oriented Analysis, Sequence Diagrams, Software Models, State Diagrams, UML.

I. Introduction

UNIFIED Modeling Language (UML) has been widely used as a standard language for modeling the software. UML 2.0 [1] consists of thirteen types of diagrams: class, composite structure, component, deployment, object, package, activity, use case, statechart, sequence, communication, interaction overview, and timing. Each diagram is dedicated to a different design aspect. Many different UML diagrams are usually involved in software development. Using more than one diagram to design a system is necessary but can leave the system in an inconsistent state and hence produce errors. Finding inconsistencies in software design before the design is implemented is very important. "Error detection and correction in the design phase can reduce total costs and time to market" [2].

A consistency problem may arise due to the fact that some aspects of the model will be described by more than one diagram. Hence, we should pay much attention to the consistency in the early phases of the system development and it is important that the consistency of a system should be checked before implementing it [3]. To avoid such errors, we should check the consistency among the diagrams and make sure that the diagrams are consistent.

Mohammad N. Alanazi is a PhD Student at Computing and Information Science Department, Kansas State University, Manhattan, KS 66506, USA (e-mail: alanazi@ksu.edu).

David A. Gustafson is a professor at Computing and Information Science Department, Kansas State University, Manhattan, KS 66506, USA (e-mail: dag@ksu.edu).

Many researchers found that the problem of ensuring consistency between UML diagrams has not been solved yet [4]. The UML specification does not enforce many 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-based intraagent (or intraobject) behavioral descriptions with scenario-based interagent (interobject) descriptions has recently focused much interest among the software engineering community" [5]. Identifying inconsistencies between UML diagrams can help the developers to find errors and fix them at early stages. Furthermore, current UML CASE-tools (e.g. Rational Rose) provide poor support for maintaining consistency between UML diagrams. So, helping to solve this problem can make a great contribution to the software development process.

This work proposes a new approach to discover the inconsistency in multiple state diagrams not just in a single state diagram. The approach analyzes multiple state diagrams. In our research, we discovered that there is essential information about the relationships between transitions in different state diagrams that is not captured in any UML diagram. This information is critical to understanding the specified system. In our approach this information is in the transition set which must be provided by the developer. The transition set includes all legal transitions that are allowed in the system. This set pairs transitions in multiple state diagrams together.

II. The Super State

Our approach for consistency analysis combines the state information of multiple state diagrams into a composite super state, SS. The super state has the form [s_1, s_2, …, s_n] where s_i is the state of object i and n is the total number of objects. This super state 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 provides the complete collaborative view of a set of objects in the model.

SS is changed after each message call. For every call we have <SS_{pre}, *transition*, SS_{post}> where SS_{pre} is the super state before *transition* and SS_{post} is the super state after the *transition* has been taken. In SS_{post}, only the state of one object is changed. This object must be the destination object of the message call. The state of the other objects remains in the same state as they were before call. We calculate the super state of multiple state diagrams after each valid transition and that is used to evaluate each sequence diagram. A valid sequence diagram should be a subsequence of the set of sequences that are possible in a super state. Invalid and impossible sequences can be identified.

III. The Approach

The information in UML diagrams are related to each other and represent different views of a system. Hence, they can be validated against each other. Given a statechart diagram, researchers [3] have shown how to validate it against a sequence diagram. On the other hand, given a sequence diagram, it can be validated against a statechart diagram [6, 7].

However, we are proposing a new approach to check the consistency between multiple state diagrams and one or more sequence diagrams. Our analysis, the *Super State Analysis (SSA)*, focuses on multiple state diagrams instead of a single state diagram.

The diagram on Fig. 1 shows the complete analysis process and the relationships between the different sources of information. Some information is known from the domain knowledge and provided by the developer while some other information is extracted from the existing information and generated automatically. *SSA* uses the provided information to generate some information automatically. Comparing the information from different sources allows us to detect the inconsistencies. *SSA* includes some inconsistencies that can be detected by the computer and some other faults that can be identified by the human. *SSA* performs five types of comparisons to detect the inconsistencies.

The diagram on Fig. 1 includes the 12 information sets that are involved in *SSA* model. The system developer provides the UML state diagrams, the transition set and UML sequence diagrams (D1, D2, and D3). The developer identifies the necessary super states, impossible super states, necessary single step transitions, and the impossible single step transitions (H1, H2, H3, and H4). The *SSA* tool is automatically generates three big sets: set of all generated super states, set of all valid single step transitions, and set of all generated sequences (T1, T2, and T3). These three sets are generated using the UML state diagrams and the provided transition set. The valid sequences (S) are extracted from the UML sequence diagram. Table I describes each component involved in the analysis and the source of each.

The *SSA* tool uses the UML state diagram (D1) and the transition set (D2) to generate the set of all generated Super States (T1). Also, the tool uses the transition set (D2) to compute the set of all generated sequences (T3). Moreover, the tool uses the transition set to compute the set of all valid single step transitions (T2). The developer uses the domain knowledge to indentify the necessary super states, impossible super states, necessary single step transitions, and impossible single step transitions. Furthermore, the UML sequence diagram is used to extract the sequences which will compare to the set of all generated sequences.

IV. Comparisons

The Super State Analysis consists of five types of comparisons to detect the inconsistencies in the multiple state diagrams and sequence diagrams.

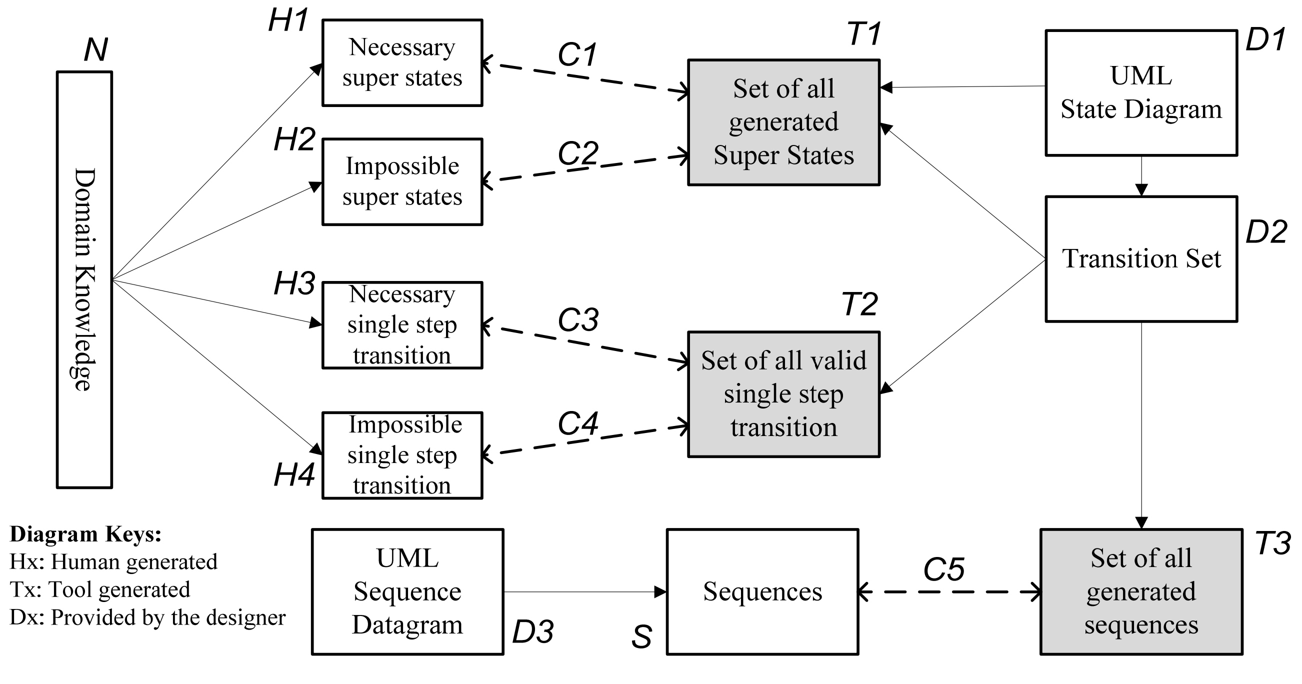


Fig. 1 *SSA* Model

TABLE I
DESCRIPTION OF EACH COMPONENT INVOLVED IN *SSA* MODEL

Box	Name	Description	Source
N	Domain Knowledge	The facts that are known by the developer of the system	Known from the domain knowledge
H1	Necessary Super States	The set of states that are identified to be necessary super states.	Domain Knowledge
H2	Impossible Super States	The set of states that are identified to be impossible super states.	Domain Knowledge
H3	Necessary single step transitions	The set of transitions that are identified to be necessary single step transitions	Domain Knowledge
H4	Impossible single step transitions	The set of transitions that are identified to be impossible single step transitions	Domain Knowledge
T1	Set of all generated Super States	These super states are generated automatically using the UML diagram and transition set	Generated Automatically by the Tool
T2	Set of all single step transition	This set contains all of the single step transitions. These transitions are generated automatically using the transition set	Generated Automatically by the Tool
T3	Set of all generated sequences	This set contains all of the legal sequences that are allowed by the system. This set is generated automatically using the transition set	Generated Automatically by the Tool
D1	UML State Diagram	The state diagrams that are written by the developer who specifies the system	Developer
D2	Transition Set	The set of all legal transitions that are allowed by the system. The developer provides this set	Developer
D3	UML Sequence Datagram	The sequence diagrams that are written by the developer who specifies the system	Developer
S	Sequences	Sequences that are extracted from the UML sequence diagrams	Generated Automatically by the Tool

1) C1: Compares the set of all generated super states (T1) with the set of necessary super states (H1).
2) C2: Compares the set of all generated super states (T1) with the set of impossible super states (H2).
3) C3: Compares the set of all valid single step transitions (T2) with the set of necessary single step transitions (H3).
4) C4: Compares the set of all valid single step transitions (T2) with the set of impossible single step transitions (H4).
5) C5: Compares the set of all generated sequences (T3) with the set of sequences (S) which are extracted from the provided UML sequence diagrams.

C1 and C2 detect the valid and invalid super states while C3 and C4 identify the illegal and missing transitions. C5 detects the invalid sequences. This comparison is fully automated since both T3 and S are generated automatically. The other four comparisons can be automated if we formalize the four sets: H1, H2, H3, and H4 and feed them to the system. By comparing these four sets to the generated sets: T1 and T2 the inconstancies can be detected automatically.

V. ERROR DISCOVERY

Super State Analysis (*SSA*) discovers inconsistencies in super states, single step transitions, and sequences.

A. States Inconsistencies

The valid and invalid states will possibly be identified by *SSA*. If a Super State (SS) is generated by Box T1, but it is not in set of valid states (Box H1) then the state is invalid SS. This could happen if there is a wrong transition in the transition set. On the other hand, if a Super State is in the set of valid states (Box H1), but it is not generated by Box T1, then this SS is a valid super state and should be generated. SS wouldn't be generated if there is a missing transition in the transition set or in the state diagram.

The following kinds of inconsistencies can be discovered by this analysis:

1) Impossible super states
2) Unreachable super states

B. Single Step Transitions Inconsistencies

The necessary and impossible single step transitions (Box H3 and Box H4) are known from the domain knowledge. The set of all valid single step transitions (Box T2) are generated automatically using the transition set. Comparing those sets will discover some legal and illegal transitions.

If a necessary transition does not appear in set of all valid single step transitions that means this necessary transition is missing. Furthermore, if an impossible transition appears in the set of all valid single step transitions that means this transition is illegal.

The following kinds of inconsistencies are discovered by this analysis:

1) Illegal transitions
2) Missing transitions

C. *Sequences Inconsistencies*

The tool generates the sequences using the transition matrix. To validate a UML sequence diagram, the tool extracts the sequences first (Box S), then, compares them to the set of all generated sequences (Box T3). If there is a matching sequence in that set, this sequence is valid. Otherwise, it is an invalid sequence.

The following kinds of inconsistencies are discovered by this analysis:

1) Illegal Sequences

The tool uses the UML state diagrams and the transition set to generate the set of all generated Super States (SS). Also, the tool uses the transition set to compute the set of all generated sequences. Moreover, the tool uses the transition set to compute the Set of all single step transitions.

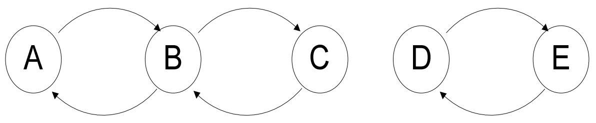
From the domain knowledge, we identify the sets of valid and invalid Super States (SS) and the necessary and impossible single step transitions.

The UML sequence diagram is used to extract the sequences which will be compared to the set of all generated sequences.

The inconsistency can be fixed by several ways. It can be fixed by adding or removing a fact to the domain knowledge. Another way to fix the inconsistencies is correcting the state diagram by adding a new transition (or removing one).

VI. The Transition Matrix

The transition matrix 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 super state (SS). Consider a program that has class X and class Y. Let class X has an initial state A and two other states, B and C, while class Y has an initial state D and a second state E. Fig. 2-a shows the state diagram for class X and Fig. 2-b shows the state diagram for class Y. The state diagrams depict how instances of X and Y can transition between those states. Let class Y makes the transition between state D and state E whenever class X makes the transition from state A to state B. Table I shows possible transitions in the super state that is the cross-product of all states with one instance of X and one instance of Y.



a- State diagram for Class X b- State diagram for Class Y

Fig. 2 State Diagrams for Class X and Class Y

An entry in a cell in T_1 (Table II) 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, $T_1 + T_1*T_1 + T_1*T_1*T_1$ +…. 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.

TABLE II
Super State Transition Matrix T_1

T1	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

VII. The Example

In this section, we demonstrate our approach by simple library system. 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. Fig. 3 shows the class diagram for this model. Fig. 4 and Fig. 5 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 originally was created by a team of students trying to create a correct model of a simple library system.

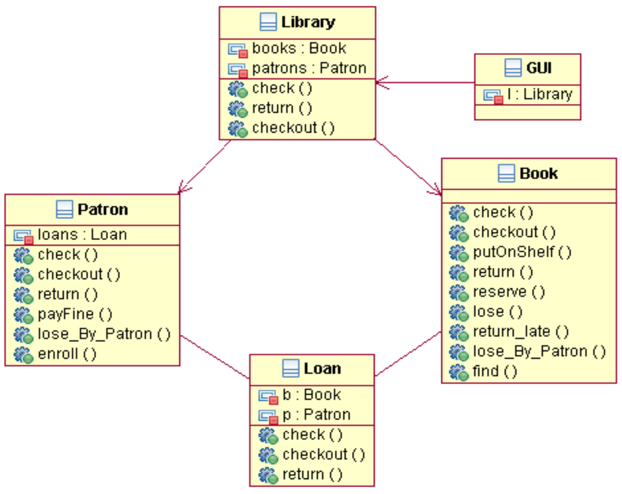


Fig. 3 The Class diagram for the Library Example

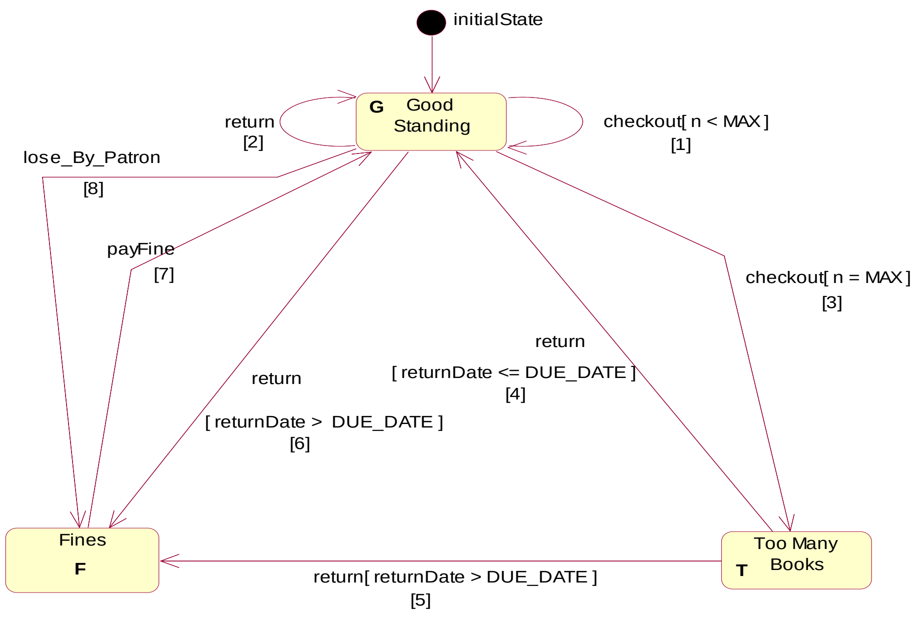


Fig. 4 The State Diagram for Patron

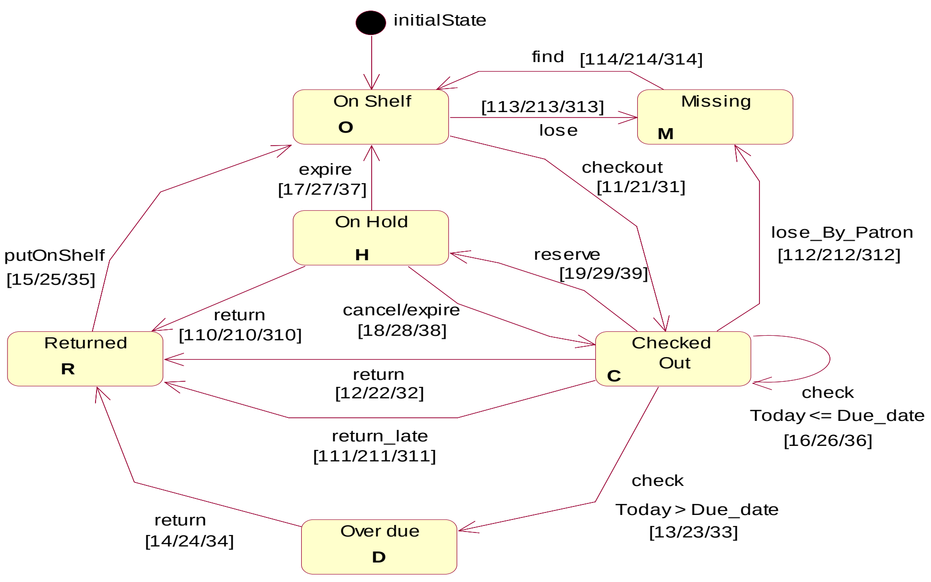


Fig. 3 The State Diagram for Book

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.

A book object has six states; *On Shelf, Missing, On Hold, Checked Out*, *Overdue*, and *Returned*. We will call these states *O, M, H, C, D*, and *R* respectively for the rest of this paper.

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 three books. We now pair the transitions from the patron and book objects that can occur together. An 'X' indicates that we are not concerned about the state of the object. The transition set is shown in Table III.

TABLE III
TRANSITION SET FOR THE LIBRARY EXAMPLE

$SS_{pre} \rightarrow SS_{post}$	Transition	Description
GOXX → GCXX	checkout[n<MAX], checkout	Check out a book (if at least one X = O || R)
GXOX → GXCX	checkout[n<MAX], checkout	
GXXO → GXXC	checkout[n<MAX], checkout	
GOXX → TCXX	checkout[n=MAX], checkout	Check out a book (if X = C || H || D)
GXOX → TXCX	checkout[n=MAX], checkout	
GXXO → TXXC	checkout[n=MAX], checkout	
GCXX → GRXX	return, return	Return book on time
GXCX → GXRX	return, return	
GXXC → GXXR	return, return	
GDXX → FRXX	return[returnDate>dueDate], return	Return an over due book
GXDX → FXRX	return[returnDate>dueDate], return	
GXXD → FXXR	return[returnDate>dueDate], return	
TCXX → GRXX	return[returnDate<=dueDate], return	Patron with MAX books returns a book on time
TXCX → GXRX	return[returnDate<=dueDate], return	
TXXC → GXXR	return[returnDate<=dueDate], return	
TDXX → FRXX	return[returnDate>dueDate], return	Patron with MAX books returns an over due book
TXDX → FXRX	return[returnDate>dueDate], return	
TXXD → FXXR	return[returnDate>dueDate], return	
GCXX → FMXX	lose_by_patron, lose_by_patron	Patron lost a book
GXCX → FXMX	lose_by_patron, lose_by_patron	
GXXC → FXXM	lose_by_patron, lose_by_patron	
GCXX → GHXX	reserve	Patron holds a book
GXCX → GXHX	reserve	
GXXC → GXXH	reserve	
TCXX → THXX	reserve	Patron with MAX books holds a book
TXCX → TXHX	reserve	
TXXC → TXXH	reserve	
GHXX → GCXX	cancel/expire	Cancel/Expiration of holding book (n < MAX)
GXHX → GXCX	cancel/expire	
GXXH → GXXC	cancel/expire	
THXX → TCXX	cancel/expire	Cancel/Expiration of holding book (n = MAX)
TXHX → TXCX	cancel/expire	
TXXH → TXXC	cancel/expire	
O → M	lose	A book lost by the library
M → O	find	A book found by the library
F → G	payFine	Patron pays fine
C → D	check[today>Due_date]	Book becomes over due
C → C	check[today<=Due_date]	Book remains checked out
R → O	putOnShelf	Book is re-shelved
H→ R	return	Return an on hold book
C→ R	Return_late	Return a late book

The initial transition matrix A_1 has column and row headings with quadruple representing the states of the four objects. For this model there are 3*6*6*6 = 648 combinations of the four objects. Table IV shows a portion of the initial transition matrix A_1.

TABLE IV
PORTION OF A_1

	GOOO	*GOCO*	*GODO*	*GORO*	*GCOO*
GOOO		1,21			1,11
GOCO		26	23	2,22	
GODO					
GORO	25				
GCOO					16

The row headings are the initial states and the column headings are the final states. The numbers in the table arise from Fig. 4 and Fig. 5. 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 three numeric identifiers for each transition since we have three instances of that object. For example, *GOOO* → *GOCO* 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 $A_1 * A_1$ which identifies all the states we can reach in two steps. Table V shows a portion of A_2.

TABLE V
PORTION OF A_2

	GOOO	*GOCO*	*GODO*
GOOO		(1,21)(26)	(1,21)(23)
GOCO	(2,22)(25)	(26)(26)	(26)(23)
GODO			
GORO		(25)(1,21)	
GCOO	(2,12)(15)		

From Table V we can observe that it is possible to go from *GOCO* to *GOOO* by first returning the second book and then shelving it.

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 three books is in *O* or *R*. Clearly the patron cannot 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 all books are in *C* or *D*. In order for the patron to be in *F,* one of the three 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 *FRCO* and *FCRO* there is no valid single step transition to *FRRO*. This means that if one book is returned late, the patron goes to *F* status and cannot return the other book until the fine is paid.

Some inconsistencies found in the library example:
1) The patron can’t return the book if she/he find it later on.
2) The patron can’t return any of her/his other books until the fine is paid first.
3) The patron can’t lose an overdue book.
4) The patron can’t lose a book if he is in state ‘T’.
5) The system reaches an invalid state when the patron checked out MAX books and trying to return a late book.

VIII. RELATED WORK

There are several different approaches that have been proposed to perform consistency checking between UML diagrams. Some approaches use transformation to convert one diagram to another [2, 4, 7, 8, 9, 10] while others detect the inconsistencies by comparing one diagram to another using consistency rules [3, 11] Moreover, many approaches use formalism, such as OCL and Z, to enforce the consistency [6, 12, 13, 14].

Almost all approaches focus on all or some of six types of UML diagrams. Namely, use case, class, object, sequence, collaboration, and statechart diagram.
[15] studies the consistency between use case, class, sequence, and statechart diagram. [4] studies the consistency between class, object, sequence, collaboration, and statechart diagram. [12] studies use case, class, sequence, and statechart diagram. [9] studies the consistency between three diagrams: class, sequence, and statechart diagram. [3, 6, 7, 8] study the consistencies between sequence and statechart diagram. [16] studies the class diagram and statechart diagram. .

The researchers pay the attention to enforce consistency between only two diagrams (e.g. single sequence diagram vs. single statechart diagram). However, our approach is unique in that we are proposing a new technique to check the consistency between multiple state diagrams and one or more sequence diagrams. Moreover, the approach focuses on multiple state diagrams instead of a single state diagram.

A. Transformation

The consistency checking in the transformational approaches is done in two steps. First, the UML diagrams are converted to interpreted diagrams. Second, the interpreted diagrams are compared to each other to detect the inconsistencies. If one diagram cannot convert to the other, then both diagrams are converted to intermediate diagrams to perform the comparison and detect the inconsistencies. These approaches require the developers to identify set of transformational rules to apply them to the diagrams in order to produce the interpreted diagrams. As a result of that, more time needs to be spent to prepare the diagrams for the comparison.

Our proposed approach converts the state diagrams to sets of valid and invalid super states. Also, it converts the transition set to sets of single step transitions and sequences. However, our approach does not require any transformational rules since the sets are generated directly from the diagrams.

Alexander Egyed [4] 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. This approach needs two sets of rules: transformation rules and consistency rules. If one diagram can't transform to another, then both diagrams transformed to an intermediate diagram to compare.

Hongyuan Wang et al.[8] propose an approach that checks the consistency between sequence diagrams and state diagrams. The approach converts statecharts using Finite State Processes and transforms sequence diagram to messages trace. They use an existing tool LTSA to support their approach. However, the approach considers only single sequence diagram and single stateschart diagram.

Wuwei Shen et al. [7] propose to build a message graph from a state chart diagram and then go through the graph based on the sequence of the messages retrieved from a sequence diagram to find any inconsistency between these two diagrams. Based on this method, a tool called ICER is developed to provide software developers with automatic consistency checking in the dynamic aspects of a model. However, the approach considers only single statechart vs. single sequence diagram.

Orest Pilskalns et al. [2] present 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.

B. *Consistency Rules*

In consistency rules approaches, the consistency is checked using the set of consistency rules. The diagrams are compared to each other directly without transformation or formalism.
Boris Litvak et al. [3] present an 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 only one object lifeline in the sequence diagram 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.

Alexander Egyed [11] introduces an approach for quickly, correctly, and automatically deciding what consistency rules to evaluate when a model changes. The approach does not require consistency rules with special annotations. Instead, it treats consistency rules as black-box entities and observes their behavior during their evaluation to identify what model elements they access. The UML/Analyzer tool integrated with Rational Rose fully implements this approach. It was used to check 24 types of consistency rules. The author found that the approach provided design feedback correctly and required, in average, less than 9 ms evaluation time per model change with a worst case of less than 2 seconds at the expense of a linearly increasing memory need.

C. *Formalism*

Since UML is not precise enough, some researchers formalize the UML diagrams to some formal languages (e.g. Z). They then compare this formalism to detect the inconsistencies between the diagrams.

Yves Dumond et al. [6] show that it is possible to integrate semi-formal and formal methods for the dynamic behavior of the UML models. The objective is to favor 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. To preserve the object notation, they name the pi-calculus processes with the name of the objects. 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.

Krishnan [13] 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 [15] describe 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. They use the formal language to ensure the consistency between two diagrams.

IX. Conclusion and Future Work

To avoid errors in UML diagrams, we should check the consistency among the diagrams and make sure that the diagrams are consistent to each other. To accomplish this, we have proposed this work to identify the problem that may arise due to the fact that some aspects of the model will be described by more than one diagram.

We proposed a solution that analyzes the multiple UML state diagrams and UML sequence diagrams to detect inconsistencies in states, single step transitions, and sequences. Our approach, The Super State Analysis (*SSA*), consider transition of multiple state diagrams instead of transition of a single state diagram. Super State Analysis generates automatically three different sets to detect the five types of inconsistencies. *SSA* generates the set of all generated super states (T1), set of all valid single step transitions (T2), and set of all generated sequences (T3). *SSA* performs five types of comparisons between these generated sets and the provided sets to detect the inconsistencies. Namely the super state analysis compares set T1 with set H1, set T1 with set H2, set T2 with set H3, set T2 with set H4, and set T3 with set S. *SSA* identifies five types of inconsistencies: impossible super states, unreachable super states, illegal transitions, missing transitions, and illegal sequences.

On the future work, we are planning to expand the case study to a bigger one with more states and sequence diagrams. We will consider more interaction between the state diagrams with different number of instantiations. Furthermore, we will investigate the different type of inconsistencies using the five comparisons. Moreover, we will use the set notations to formalize the different sets that are involved in the comparisons. Also, we plan to build the *SSA* tool to perform all five comparisons automatically. We are also developing approaches to minimize the state explosion to allow the *SSA* to scale to larger systems.

REFERENCES

[1] OMG Unified Modeling Language Specification, UML 2.0, *Object Management Group*, 2006, http://www.uml.org.

[2] O. Pilskalns, A. Andrews, S. Ghosh, & R. France, Rigorous Testing by Merging Structural and Behavioral UML Representations, *Proc. 6th Int. Conf. on UML*, San Francisco, CA, 2003, 234-248.

[3] B. Litvak, S. Tyszberowics, & A.Yehudai, Behavioral Consistency Validation of UML Diagrams, *Proc. 1st Int. Conference on Software Engineering and Formal Methods*, 2003, 118-125.

[4] A. Egyed, Scalable Consistency Checking between Diagrams-The ViewIntegra Approach, *Proc. 16th Annual International Conference on Automated Software Engineering,* 2001, 387-390.

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

[6] Y. Dumond, D. Girardet, & F. Oquendo, A relationship between sequence and statechart diagram, A Workshop, *Proc. Dynamic Behaviour in UML Models: Semantic Questions*, York, UK, 2000.

[7] W. Shen & W. Low, Consistency Checking Between Two Different Views Of a Software System, *Proc. 10th IASTED Int. Conf. on Software Engineering and Applications*, Dallas, TX, 2006.

[8] H. Wang, T. Feng, J. Zhang, & K. Zhang, Consistency check between behaviour models, *Proc. IEEE International Symposium on Communications and Information Technology*, China, 2005, 486-489.

[9] R. Straeten , J. Simmonds & V. Jonckers, Maintaining Consistency between UML Models Using Description Logic, *Journal S'erie L'objet - logiciel, base de donn'ees, r'eseaux, 10*(2-3), 2004, 231-244.

[10] R. Wagner, H. Giese, & U. Nickel, A Plug-In for Flexible and Incremental Consistency Management, *Proc. International Conference on the UML 2003*, San Francisco, October 2003, 78-85.

[11] A. Egyed, Instant consistency checking for the UML, *Proc. 28th International Conference on Software Engineering*, China, 2006, 381-390.

[12] H. Gomaa & D. Wijesekera, Consistency in Multiple-View UML Models: A Case Study, *Proc. of Workshop on Consistency Problems in UML-based Software Development, 6th Int. Conf. on the UML*, San Francisco, 2003.1-8.

[13] P. Krishnan, Consistency Checks for UML, *Proc. 7th Asia-Pacific Software Engineering Conference*, Singapore, 2000, 162-169.

[14] S. Kim & D. Carrington, A Formal Object-Oriented Approach to defining Consistency Constraints for UML Models, *Proc. of the 2004 Australian Software Engineering Conference* Australia, 2004, 87-94.

[15] L. Kuzniarz & M. Staron, Inconsistencies in Student Designs, *Proc. 2nd Workshop on Consistency Problems in UML-based Software Development*, San Francisco, CA, 2003, 9-18.

[16] Z. Pap, I. Majzik, A. Pataricza, & A. Szegi, Completeness and Consistency Analysis of UML Statechart Specifications, *Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, Hungary, April, 2001, 83-90.