Inconsistency between State and Sequence Diagram

A Thesis

By

Sakr Hamadeh

Submitted in Partial Fulfillment of the Requirements for the Degree of Masters of Science in Computer Science

Department of Computer Science
Faculty of Natural and Applied Sciences
Notre Dame University – Louaize
Zouk Mosbeh, Lebanon
June 2005

Inconsistency between State and Sequence Diagram

By

Sakr Hamadeh

Approved:

[illegible]

Dr. Omar Rifi: Assistant Professor of Computer Science
Advisor & Chairman of Committee

[illegible]

Dr. Khaldoun Khalidi: Assistant Professor of Computer Science

[illegible]

Dr. Ahmad Shahin: Assistant Professor of Computer Science

[illegible]

Dr. Ziad Rached: Assistant Professor of Mathematics

Date of Thesis Defense: April 19, 2005

ACKNOWLEDGEMNENTS

As I prepare to take my Masters Degree, there are a number of people whom I would like to take this opportunity to thank, for without their help and encouragement, I would not be at the stage where I now find myself.

The main credit for such support should undoubtedly go to my thesis advisor,Dr. Omar Rifi, whose inspiring interest and ceaseless enthusiasm for the topic explored in this thesis has meant a great deal to me. The topic itself was, indeed, a suggestion of him, and for that as well I am grateful, as it allowed me to combine the best of two worlds – my 'academic' preference for linguistics, and my still growing 'amateurism' – in the good sense of the word – in literature. I would like to express my thanks for the remarkable balance he never failed to strike between making insightful comments and useful suggestions on the one hand, and showing confident and great respect for the independent thought and style of his student – me, in this case – on the other hand.

I thank my Parents, for their countless practicalities that have helped me to deliver this work, but also, more fundamentally, for their continual support and belief in me, offering me the opportunity to study as much as I want.

Finally, I want to thank Notre Dame University and all professors and teachers who had taught me throughout the graduate study. And I would like to dedicate this thesis to all the people that have had a formative influence on me over the years – teachers in primary and secondary school, professors and research assistants, and of course my parents, other relatives, and friends. Each in their own way, they have helped to unlock some of the mysteries of this world, but also to stand in uncomprehending awe for many others; not to give up in face of difficulties ahead, but also to recognize the limits of any human being's possibilities; to enjoy the fullness of everyday existence, but also to look for ways of transcending the obvious.

ABSTRACT

UML is an acronym for Unified Modelling Language. It has become de facto the standard for the object-oriented software analysis and design stages in software development. UML is a visual modelling language, and it consists of a set of diagrams. Static diagrams are used to depict static structure of a program, whereis dynamic diagrams specify how the control flow(s) of the program should behave. The examples of behavioral diagrams are a State diagram, which describes the behavior of objects of a given class, and a Sequence diagram, which describes inter-object interactions in a given scenario.

A consistency problem may arise due to the fact that some aspects of the model may be described by more than one diagram. Hence, the consistency of the system description should be checked before implementing the system.

This thesis describes an algorithmic approach to a consistency check between UML Sequence and State diagrams. The algorithm we provide automate the validation process which handles complex state diagrams, e.g. diagrams that include forks, joins, and concurrent composite states.

TABLE OF CONTENTS

LIST OF DIAGRAMS

CHAPTER I

1- UML

Developing a model for an industrial-strength software system prior to its construction or renovation is as essential as having a blueprint for large building. Good models are essential for communication among project teams and to assure architectural soundness. As the complexity of systems increase, so does the importance of good modeling techniques. There are many additional factors of a project's success, but having a rigorous modeling language standard is one essential factor.

Since the Unified Modeling Language (UML) became the standard notation for software architecture, it has become the topic of many books, discussions, and seminars.

The Unified Modeling Language (UML) is an important tool for modeling Enterprise Application Integrations (EAI). The complex interaction among multiple applications, such as GIS, outage management (OMS), workforce management, and customer information systems, requires a tool to support many system development tasks. UML aids in modeling the business processes, managing EAI requirements, providing test criteria, and building end-user documentation. The applications that a GIS is required to interface with are constantly increasing and the applications are not only linked within a company but are also linked between companies. Traditionally, communication between the GIS and other applications has been customized, one integration at a time. The result is a set of complex and unique integration points with redundancy throughout the system. The EAI approach uses a set of tools to share data and business processes among connected applications and data stores within and, optionally, between organizations.

1.1- The history of UML

The beginnings of documented object-oriented modeling languages can be traced back to the mid-1970's when methodologists were experimenting with different approaches to object-oriented analysis and design. The number of identifiable object-oriented modeling languages increased from less than 10 to more than 50 between 1989 and 1994 (OMG, 2001). In the mid-1990's, the primary authors of the leading modeling languages (Booch, OMT, and OOSE) realized that each of their models were evolving into very similar models. In late 1995 Grady Booch, Jim Rumbaugh, and Ivar Jacobson, the primary authors of the leading modeling languages, joined forces at Rational Software Corporation and began collaborating on a unified modeling language. In early 1996 the authors released the UML 0.9 and requested feedback from the general community. The feedback they received was incorporated into the model and the authors then released the UML 0.91 document. During 1996 the Object Management Group (OMG) issued a Request for Proposals for a definition of a modeling language standard, which proved to be a catalyst for interested parties and organizations to join forces and provide a response. The History of the UML Partners consortium, established by Rational, provided a response to the RFP. The consortium included the "three amigos" (Booch, Rumbaugh, and Jacobson) and organizations such as Digital Equipment Corp., HP, IBM, MCI System-house, Microsoft, Oracle, Rational, TI, and Unisys. The UML Partners focused on improving the UML 0.91 architecture so that it met the demands of all mainstream methods and ensuring that it was general purpose in nature. The UML Partners submitted their initial UML proposal (UML 1.0) to the OMG in January 1997 and the final proposal to the OMG in September 1997, which the OMG adopted as its object modeling standard.
The OMG and the UML Partners consortium recognized that the UML was an evolving language that would need to be reviewed and evolved on a continual

basis. The UML Partners have followed standard software development practices by planning and following a revision schedule for the UML. In September 2001, the OMG released a minor revision, UML 1.4. The first major revision since UML was adopted by the OMG in 1997 is scheduled for release in late 2002.

1.2- UML Designation

The Unified Modeling Language (UML) is a simple and extensible modeling language. The UML is implementation independent in that it can be applied to any programming language, technology, and domain. The UML is supported by a number of tools by independent software vendors. The UML is process independent. It does not dictate nor require that a particular process is followed. With that said, the authors of the UML do encourage users to follow a use case driven, architecture-centric, iterative and incremental process that is object oriented and component based. Many current system design processes build upon such a framework. The UML is scalable; it can provide benefit and value to projects ranging from small-business application development to multi-national enterprise-wide application integration projects.

The UML is not a visual programming language; it is a visual modeling language. It is not simply a notation but a robust language for capturing knowledge and expressing knowledge about the system under design. During its short lifetime, the UML has become the industry-standard language for system modeling.

The UML is not a process; in fact the UML is process independent. A process should be tailored to the organization, the culture, and the problem domain at hand. Organizations will benefit from the UML as it provides a common modeling language but it does not require a common process.

value, desired completion speed, and cost. Then make appropriate judgments about which processes need to be refined with EAI technologies.

For example, perhaps timecard submissions have high frequency, low value, must be completed every week, and cost one hour of each employee's time per week. Automated crew dispatch to a serious SCADA event may occur once a week, affect hundreds of customers, must be completed as quickly as possible, and can cost thousands of dollars per minute in lost revenue. The latter may be considered more meritorious for EAI streamlining that the former. This management level decision is assisted with simple diagrams that show the differences between as-is and to-be (or "could-be") business processes.

Comparing as-is to to-be diagrams provides a powerful visual tool that encourages management support of a project and provides valuable assistance during early EAI acquisition phases. Top-down management support of EAI initiatives is imperative, since the owners of a company's data stores are justifiably hesitant to allow outside influences on their data. UML diagrams help with this support.

The key to supporting business processes is the movement and manipulation of data within the enterprise. Writers of use case diagrams should keep this goal in mind for the lowest-level use cases. After the use case diagrams are completed, additional information is available to help decide on appropriate infusions of EAI technology. Such technology should facilitate data movement in the service of important business processes. Eventually, EAI developers will need the use case diagrams and sequence diagrams to develop and test the pieces that move data between components.

It's clear from this discussion that UML provides important benefits to a company's EAI plans. Management needs simple visual tools to help make important acquisition decisions and decide which business processes deserve EAI attention. Developers need it to help with implementation and testing. Users need it to cooperate with developers during early design phases of the project. Everyone benefits from UML's common, understandable language.

In other words, the benefits of UML can be listed as:

-Since system design comes first, UML enables re-usable code to be easily identified and coded with the highest efficiency, thus reducing software development costs

-UML enables logic 'holes' to be spotted in design drawings so that software will behave as expected

-The overall system design described in UML will dictate the way the software is developed so that the right decisions are made early on in the process. Again, this reduces software development costs by eliminating re-work in all areas of the life cycle.

-UML provides an enterprise level view of the system and, as a result, more memory and processor efficient systems can be designed

-UML enables ease of maintenance by providing more effective visual representations of the system. Consequently, maintenance costs are reduced.

-UML diagrams assist in providing efficient training to new members of the development team member

-UML provides a vehicle of communication with both internal and external stakeholders as it documents the system much more efficiently

Taken and modified from: www.gita.org

Chapter II

This chapter mainly discusses the UML set of diagrams.

Introduction

UML diagrams provide many perspectives of a system during analysis and development. A complex system can be most effectively understood and therefore developed by understanding it from many angles. Let's look at a simple analogy to system modeling: you plan to build the house of your dreams. You meet with an architect to explain the features that will make this house perfect. The architect then sketches a drawing of your dream house. And voila, you are now looking at your perfect home. It is a beautiful house; the outside features include large windows, a huge red front door, and the yard is full of quaking aspens. You are looking at the house of your dreams; you write a check and tell the architect to build it. How realistic is this? What about the floor plan, the wiring plan, the window and door plan, etc.? Would you write a check at this point? My guess is you wouldn't.

Developing the many views of a system is as critical to the success of system development as are the detailed blueprints of a house plan. The UML defines a set of graphical diagrams that are used for many planning, design and implementation tasks depending on the angle that you are viewing the system. The major views of the system that UML supports are:

1) The user view
2) The structural view,
3) The behavioral view, and
4) The implementation view.

One or more diagrams for each view is defined by the UML and each provide a unique window into the system. At this point it is important to point out that few projects will utilize all the available diagrams. The UML diagrams are to be treated as a set of resources for system modeling and take care to utilize only those diagrams that provide a useful and beneficial view of your system. As you gain familiarity with the UML diagrams, it will become apparent that some diagrams are more critical than others for system modeling depending on the size of your project. The process your organization follows will determine the order in which the diagrams are created; however, the general order is: 1) use case diagrams, 2) structural and behavioral diagrams concurrently, 3) component diagram, and 4) deployment diagram.

Using a sound modeling language is essential for good communication among project teams and to provide an architectural reference to be followed through the life of a project and during future system upgrades. As systems increase in complexity, visualization and modeling become even more critical. Such is the case with EAI. An EAI project will often be comprised of specialized teams each with an expertise in GIS or OMS or CIS or middleware or business processing and the list goes on. The major benefits of using UML is it provides a common language between the teams and it provides a central repository for the EAI analysis, design, development, and implementation project plan.

The UML graphical diagrams are grouped according to their view -- user, structural, behavioral and implementation -- of the system and the diagrams of each are described in the following sections.

- ***Static diagrams*** are used to depict static structure of a program, where is
- ***Dynamic diagrams*** specify how the control flow(s) of the program should behave

1- Static diagrams

Static diagrams are Class diagram, Object diagram, Physical Diagram (Component diagram and Deployment diagram).

1.1- A Class diagram shows classes and types, their internal structure, and their interrelationship to other classes or types. Examples of such relationships are inheritance, association, aggregation, and (template class) instantiation relationships.

Classes are composed of three things: a name, attributes, and operations. Below is an example of a class.

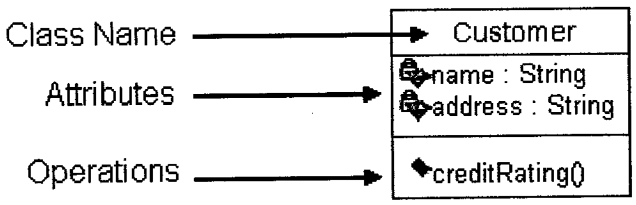

Figure 1.1 – Class Diagram

Below is an example of an associative relationship:

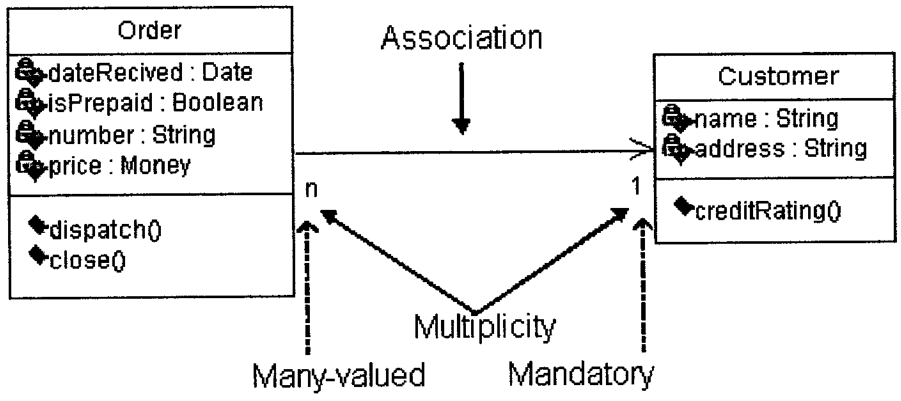

Figure 1.2 – Association relationship Diagram

The association relationship is the most common relationship in a class diagram. The association shows the relationship between instances of classes. For example, the class Order is associated with the class Customer. The multiplicity of the association denotes the number of objects that can participate in the relationship. For example, an Order object can be associated to only one customer, but a customer can be associated to many orders.

Another common relationship in class diagrams is a generalization. A generalization is used when two classes are similar having some differences. Consider the generalization below:

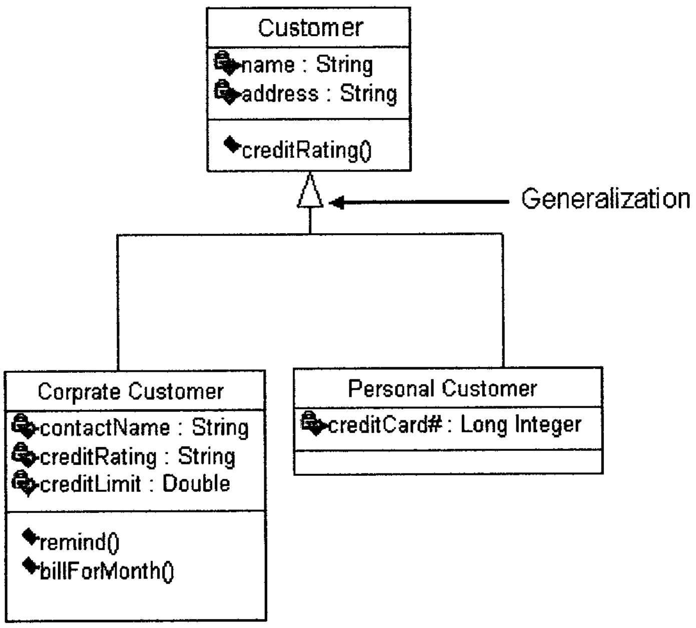

Figure 1.3 – Generalization Diagram

In this example the classes Corporate Customer and Personal Customer have some similarities such as name and address, but each class has some of its own attributes and operations. The class Customer is a general form of both the Corporate Customer and Personal Customer classes. This allows the designer to just use the Customer class for modules and do not require in-depth representation of each type of customer.

1.2- An Object diagram shows a snapshot of the detailed state of a system at a given point of time. This is an instance of a class diagram. Its use is fairly limited, mainly to exemplify data structures.

1.3- Physical Diagrams are two types: **deployment diagrams** and **component diagrams.** Many times the deployment and component diagrams are combined into one physical diagram. A combined deployment and component diagram combines the features of both diagrams into one diagram.

a- Component diagram shows the dependencies among software components. This usually means showing compilation dependencies between different binary code files, and the mapping between the source code files and the binary code files.

b- Deployment diagram shows the configuration of run-time processing elements and the software components, processors, and objects that execute on them. Software component instances represent run-time manifestation of software code units.

The combined deployment and component diagram below gives a high level physical description of the completed system. The diagram shows two nodes which represent two machines communicating through TCP/IP. Component2 is dependant on component1, so changes to component 2 could affect component1. The diagram also depicts component3 interfacing with component1. This diagram gives the reader a quick overall view of the entire system.

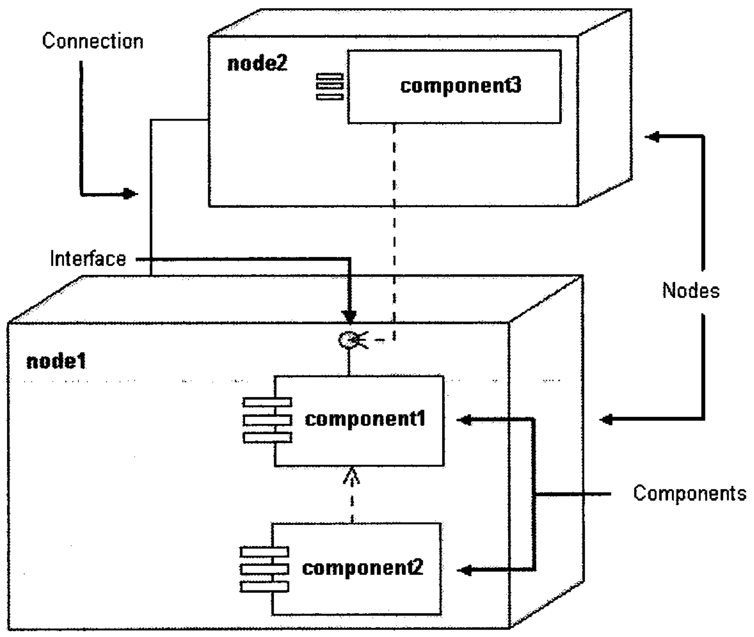

Figure 1.4 – Deployment Diagram

2- Dynamic diagrams

Dynamic diagrams are Use case diagram, Activity diagram, State diagram and Interaction diagrams (Sequence and Collaboration Diagrams).

2.1- A Use case diagram is a set of scenarios that describes an interaction between a user and a system. A use case diagram displays the relationship among actors and use cases. The two main components of a use case diagram are use cases and actors.

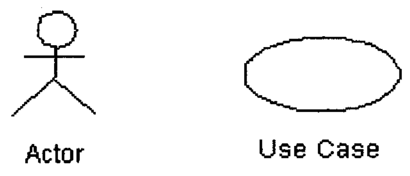

Figure 2.1 – Use case diagram

An actor represents a user or another system that will interact with the system we are modeling. A use case is an external view of the system that represents some action the user might perform in order to complete a task.

2.2- An Activity diagram describes the workflow behavior of a system. Activity diagrams are similar to state diagrams because activities are the state of doing something. The diagrams describe the state of activities by showing the sequence of activities performed. Activity diagrams can show activities that are conditional or parallel.

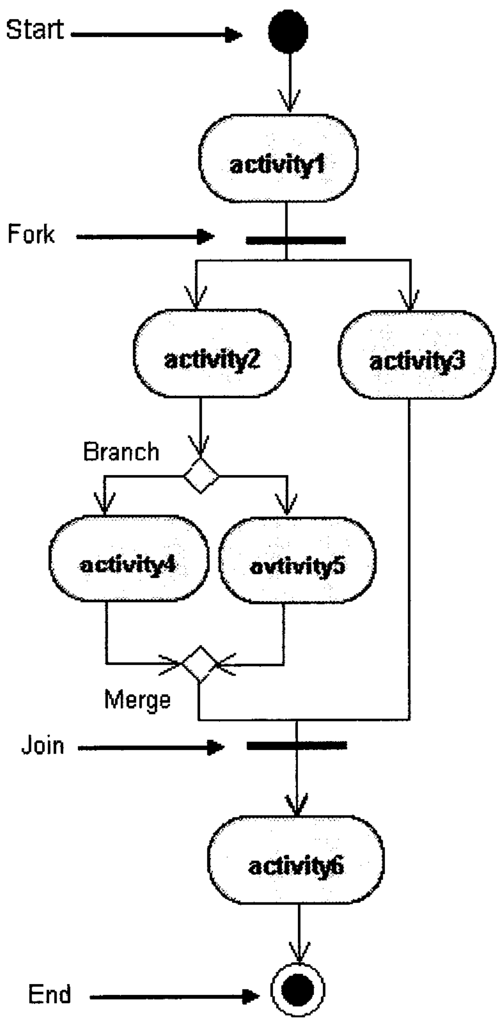

Figure 2.2– Activity diagram

Activity Diagrams are read from top to bottom and have branches and forks to describe conditions and parallel activities. A fork is used when multiple activities are occurring at the same time. The diagram below shows a fork after activity1. This indicates that both activity2 and activity3 are occurring at the same time. After activity2 there is a branch. The branch describes what activities will take place based on a set of conditions. All branches at some point are followed by a merge to indicate the end of the conditional behavior started by that branch. After the merge all of the parallel activities must be combined by a join before transitioning into the final activity state.

2.3- A State diagram is used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and tracks the different states of its objects through the system.

State diagrams have very few elements. The basic elements are rounded boxes representing the state of the object and arrows indicating the transition to the next state. The activity section of the state symbol depicts what activities the object will be doing while it is in that state.

All state diagrams being with an initial state of the object. This is the state of the object when it is created. After the initial state the object begins changing states. Conditions based on the activities can determine what the next state the object transitions to.

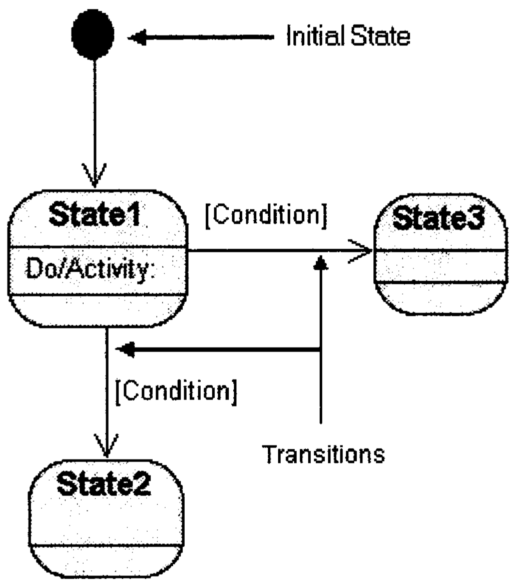

Figure 2.3 – State diagram

2.4- Interaction diagrams Interaction diagrams model the behavior of use cases by describing the way groups of objects interact to complete the task. The two kinds of interaction diagrams are **sequence** and **collaboration** diagrams.

Sequence diagrams, collaboration diagrams, or both can be used to demonstrate the interaction of objects in a use case. Sequence diagrams generally show the sequence of events that occur. Collaboration diagrams demonstrate how objects are statically connected. Both diagrams are relatively simple to draw and contain similar elements.

a- The Sequence diagram Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass. The diagrams are read left to right and descending. The example below shows an object of class 1 start a behavior by sending a message to an object of class 2. Messages pass between the different objects until the object of class 1 receives the final message.

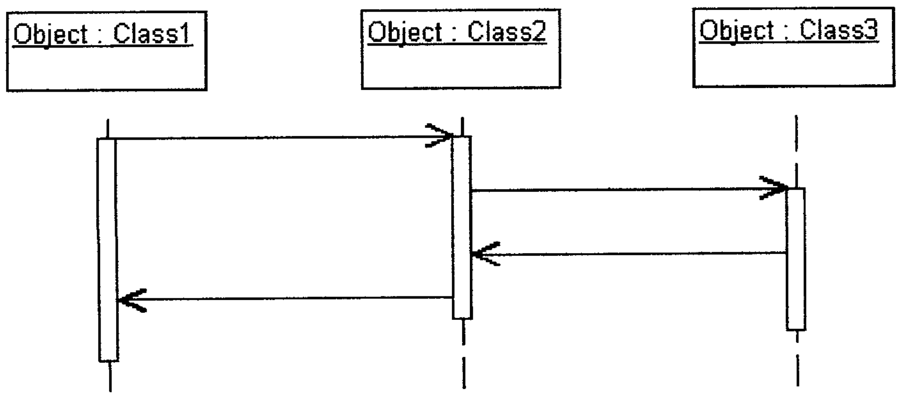

Figure 2.4– Sequence diagram

b- Collaboration diagrams are also relatively easy to draw. They show the relationship between objects and the order of messages passed between them. The objects are listed as icons and arrows indicate the messages being passed between them. The numbers next to the messages are called sequence numbers. As the name suggests, they show the sequence of the messages as they are passed between the objects.
The example below shows a simple collaboration diagram for placing an order use case.

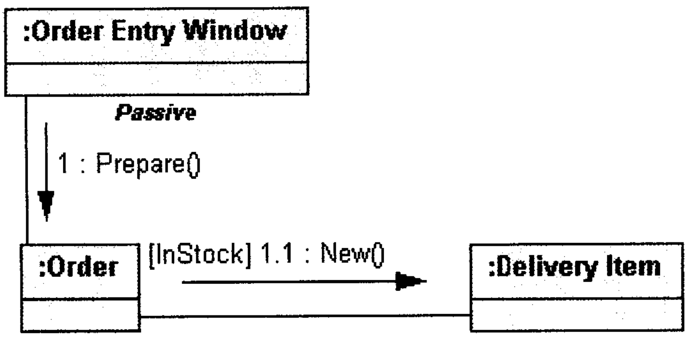

Figure 2.5– Collaboration diagram

Taken from: www.omg.org

Chapter III

Chapter III mainly discuss the following Topics: the Problem Description, then gives an Simple Example of this Problem, general Definitions to make the solution easier to be understood, Solution Description, a Run Example, the Run Results and finally Conclusions.

1- Problem Description

As the complexity of systems -- whether new or refurbished -- increases in scope and scale, so does the importance of good modeling techniques are needed. The UML meets the modeling needs of such complex systems but when these systems become more and more complicated, the integrity of a software system design become very difficult to be discovered. Further more, because UML itself is a very expressive and rich language, sometimes the model gives behaviors not expected by the designers and those behaviors could cause serious bugs for the system. For this reason, I discuss below a problem which may occurre in complex systems modeled in UML.

UML provides several kinds of diagrams to model the behavior and structure of a system under development. As mentioned before, a consistency problem may arise due to the fact that some aspects of the model may be described by more than one diagram. Hence, the consistency of the system description should be checked before implementing the system. E.g. consistencies check between UML Sequence and State Diagrams.

Let me describe more complex inconsistencies. Take, for example, some object in Sequence diagram. The outgoing legs usually occur due to some incoming leg, such as a function call or a constructor call. **The State diagram for the object's class describes the object behavior. The incoming leg to the object in the Sequence diagram serves as a trigger of the associated State diagram transition. If there is no transition for such a Trigger in the State diagram, obviously either the object behavior definition in the State Diagram is wrong or incomplete, or the Sequence diagram is incorrect. Either way, it is an inconsistency between the diagrams.**

- **The *Sequence transition* must match at least one *State transition* that emanates from a *current object state.* Otherwise the diagrams are inconsistent.**

The problem of finding such inconsistencies becomes very complex in many cases. For example, the *current object state* may consist of many simple states due to transitions into concurrent composite. These simple (non-composite persistent) states may be in different levels of State diagram hierarchy. States that change the control behavior, such as choice states and history states, make it harder to locate the *current object state* after a given transition. Many additional elements, such as transition guards (conditions), on entry on exit actions, transitions that pass hierarchical levels in the State diagram, etc, were not mentioned in this introduction.

2- A Simple Example of the Problem

The example describes the interaction between the components of a cellular phone, and the phone user. One use case of the cellular system is dialing a number. In one of the scenarios of this use case the user has entered a phone number. Just before he presses the send button, there is an incoming call. The cellular phone allows the user to answer an incoming call even after he has entered several digits. When the button, this means that the Dialer object either has to make a call or to receive an incoming call. The decision is made according to the Dialer state, whether there was an incoming call or not.

The inconsistent Sequence and State diagrams for the Dialer are specified in Figures 1 and 2, respectively. The Sequence diagram designer intended to make a call, even though the pressing of the Send button should- according to the State diagram of the Dialer- answer an incoming call. The run of the Dialer will detect the inconsistency at the first send message (message#6 in the Sequence diagram).

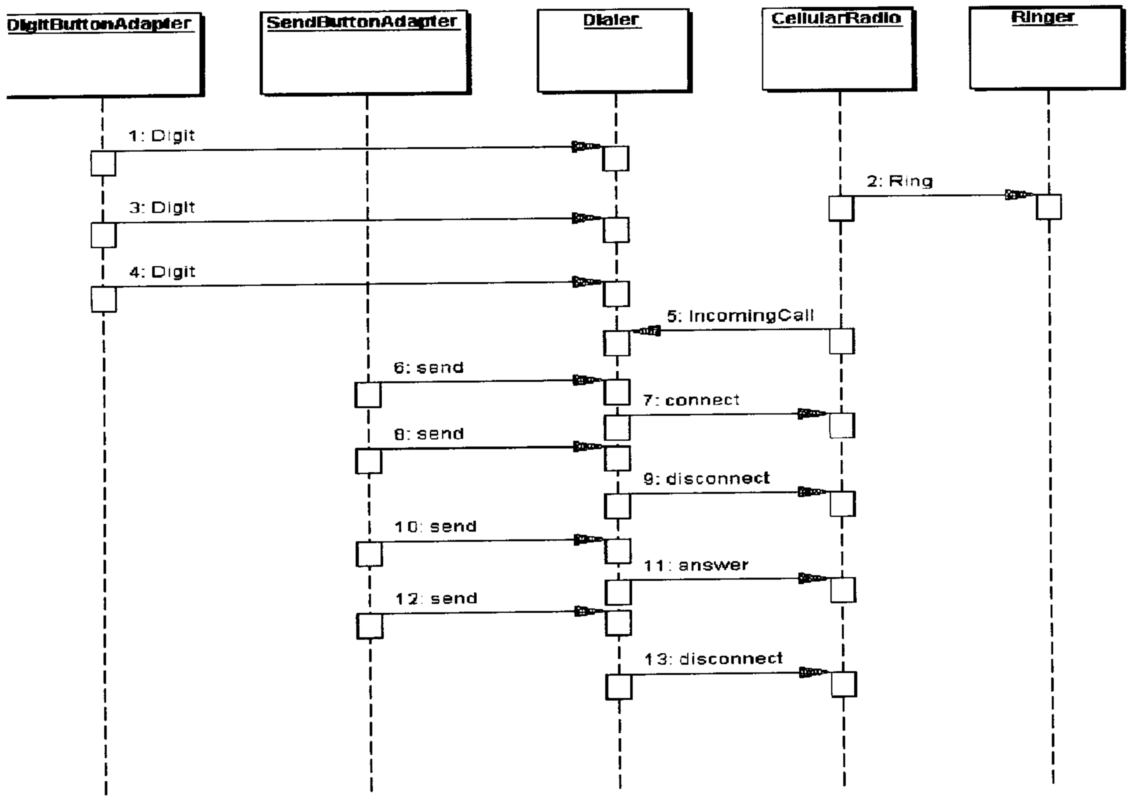

Figure 2.1 – Dialer Sequence Diagram

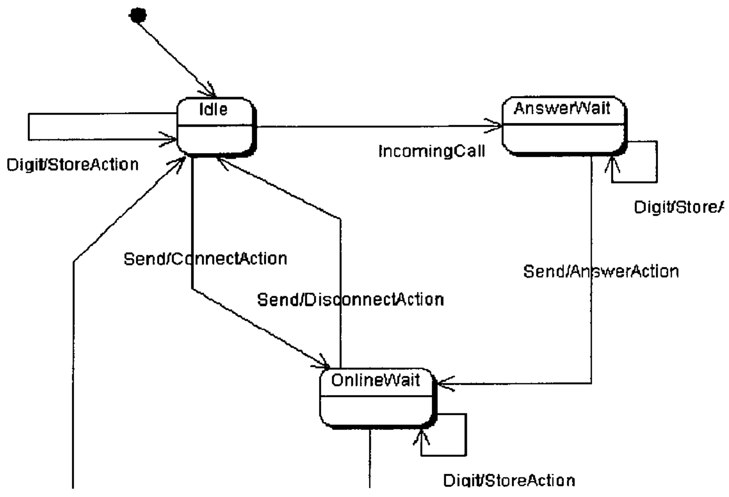

*Figure 2.2 – **Dialer State Diagram***

3- Definitions

In this section we define the terms that are used through the entire text. Some of the definitions are identical to the standard UML definitions. We also rephrased some UML definitions for easier understanding. Some definitions are non-UML ones.

3.1 Basic Behavior definitions

Event: Either a signal emission (in real-time environments) or a function call. Defined in UML as "The specification of noteworthy occurrence that has a location in time and space".

Action: An outgoing event, i.e. an event that is triggered by the associated transition.

Trigger: A trigger is an event that initiates an associated transition. We use the same term both in the Sequence diagram and in the State diagram. In the standard UML
State diagram it is named *trigger* as well, whilst in the standard UML Sequence diagram it is called *stimulus*.

Guard: A condition that must be satisfied in order to enable the associated transition to be performed.

Leg: A basic component of a Sequence diagram or a State diagram, depicted in UML as an arrow with an optional guard and an optional action. In UML, a *Leg* is called a *simple transition* in a State diagram, whereas in a Sequence diagram it is called a *message*.

Transition: Consists of a trigger and a nonnegative number of legs.

Step: An execution of transition actions, when the trigger occurs and the guards are satisfied.

Run: The Sequence of steps that are executed from the initial step to the last one.

3.2- Sequence Diagram definitions

Object: An instance that is instantiated from a class.

Sequence (diagram) transition: A transition in a Sequence diagram that consists of an incoming leg (message) and all the outgoing legs (messages), if any, triggered by it.

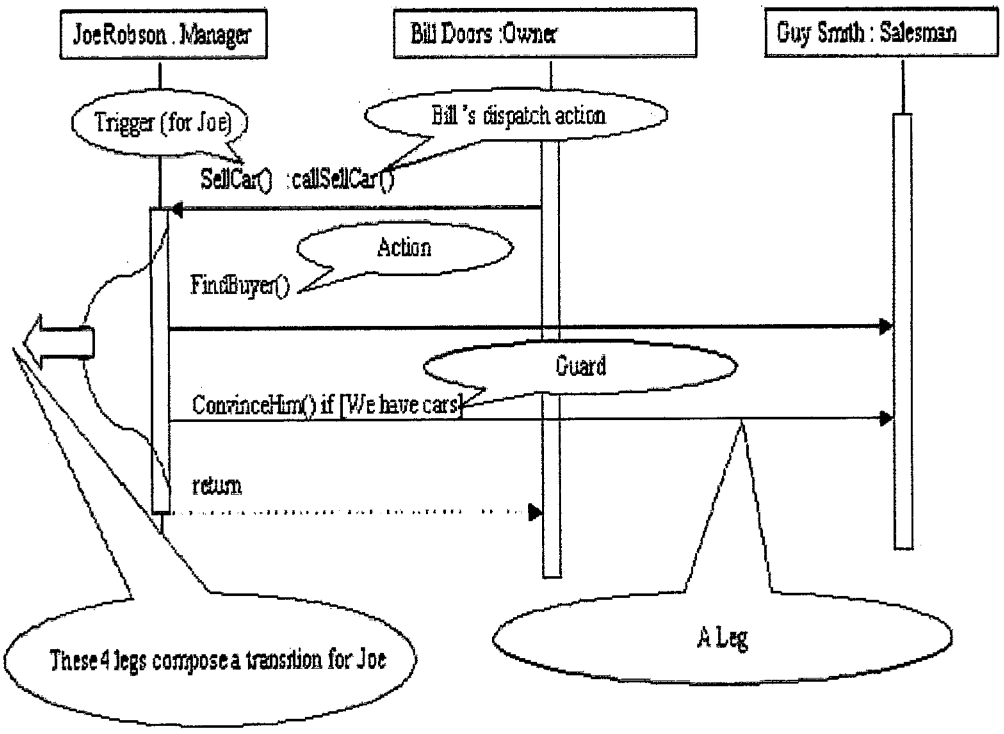

Figure 3.2 – Sequence Diagram Definitions Example

3.3- State Diagram definitions

State: A situation during the life of an object during which it satisfies some condition or waits for some event.

Pseudo state: An abstraction that encompasses different types of transient vertices in the same state machine graph.

Persistent state: A state where a thread of control stops till the end of one step.

Transient state: A state that is not a persistent state. This is equivalent to Pseudo State.

Persistent-state leg: A leg that emanates from a persistent state. It always includes a trigger.

Transient-state leg: A leg that emanates from a transient state.

Substate: A state which is composed in some composite state.

Simple state: A persistent state that does not have substates.

Fork State: A transient state that serves to split the incoming leg into two or more legs terminating on orthogonal target vertices, which are substates in the same object state.

Join state: A transient state that serves to merge several legs emanating from source vertices in different orthogonal vertices.

Shallow history: A shorthand notation for a transient state, which represents the most recent active configuration of a composite state that directly contains this shallow history
state.

Final State: A special kind of persistent state signifying that the enclosing composite state is completed. Defined as a transient state in UML.

Choice state: A state which, when reached, result in dynamic evaluation of guards on its outgoing legs.

Initial State: A transient state, which is a source for a single leg emanating towards the *default* substate of a composite state.

State (diagram) transition: A transition in a State diagram that includes the trigger and all the legs on the path between two persistent states, with no persistent states in between.

Composite State: A state that consists of concurrent (orthogonal) substates or of sequential (disjoint) substates.

Concurrent Composite State: A state that consists of concurrent (orthogonal) substates.

Exception: A leg from some state, which is located in a concurrent state, to some other state outside that concurrent state.

Saturateable state: A state that holds a count of the number of control flows that reached it. This count is called a *s*aturateable count. Fork, Composite and Concurrent composite states are saturateable states. The count is used later by the algorithm to know whether the execution should proceed beyond the saturateable state or not.

Current object state: The conjunction of all the active states in a State diagram, that defines the behavior of the object for any incoming event. At the beginning of the object lifetime, its *current object state* includes only all the initial State diagram states. The *current object state* is updated with each state transition. It can include multiple simple states, if the diagram contains concurrent composite or fork states.

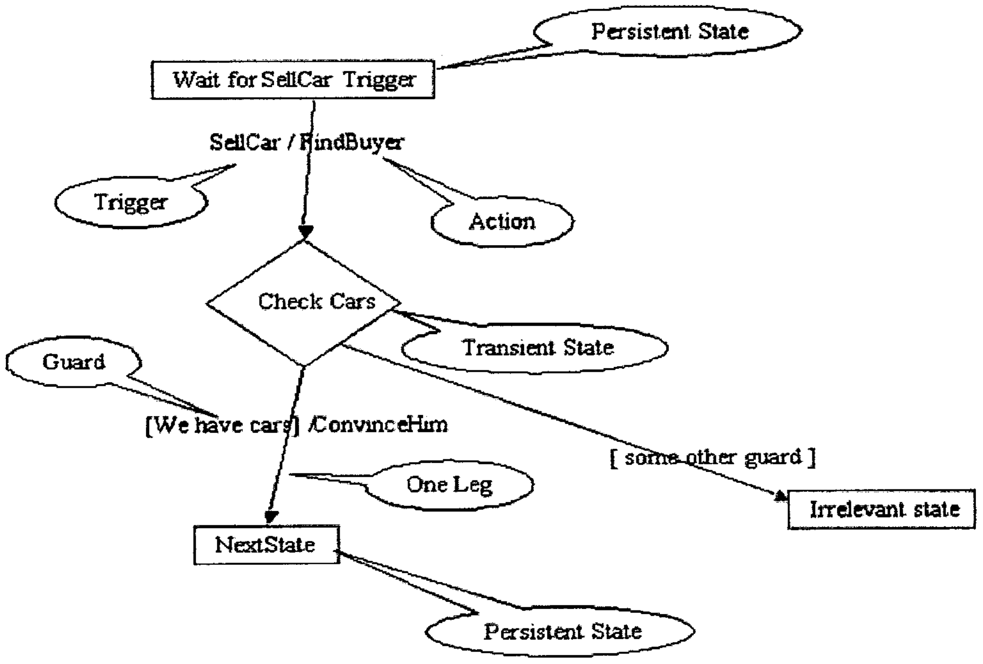

Figure 3.3 – State Diagram Definitions Example

Note that:

SellCar / FindBuyer are Persistent-State Legs. These legs always include triggers.

ConvinceThem is a Transient-State Leg

NextState is a final State and a Persistent (Composite) state

4- Solution Description

This Part describes the inputs, outputs, and the way of operation of the Algorithm below.

4.1- Inputs

This section describes the algorithm that we used. The inputs to the algorithm are:

- The name of the Sequence diagram to be checked.
- The name of the object in the Sequence diagram one would like to check.

Each run of the Algorithm validates the consistency on a specific object in a Sequence diagram. The Algorithm should be activated several times for different objects of the same Sequence diagram. This way one can ensure that some interactions between objects in the Sequence diagram are possible, according to the corresponding state diagrams of the objects, whereas other interactions are not. It is especially vital when one wants to ensure that some action always happens before another.

4.2- The main idea of the solution

One can visualize the Algorithm as a hybrid Sequence-state diagram. From the input we know which State diagram we want to check and which object in the Sequence diagram we are about to check. Given a message in the Sequence diagram, it is interpreted in one of two ways in the State diagram:

- It should be a leg leaving persistent state, if the message is an incoming message to the checked object in the Sequence diagram.

• It is a leg leaving any state, if the message is an outgoing message from the checked object in the Sequence diagram. Usually this leg will emanate from a transient state *(see State diagram definition P13)*

A message can change the *current object state* of the state diagram. Notice that the *current object state* changes according to the previous transitions.
A simple example of a coffee machine, where only a trigger changes the *current object state*, is shown in Figure 3.

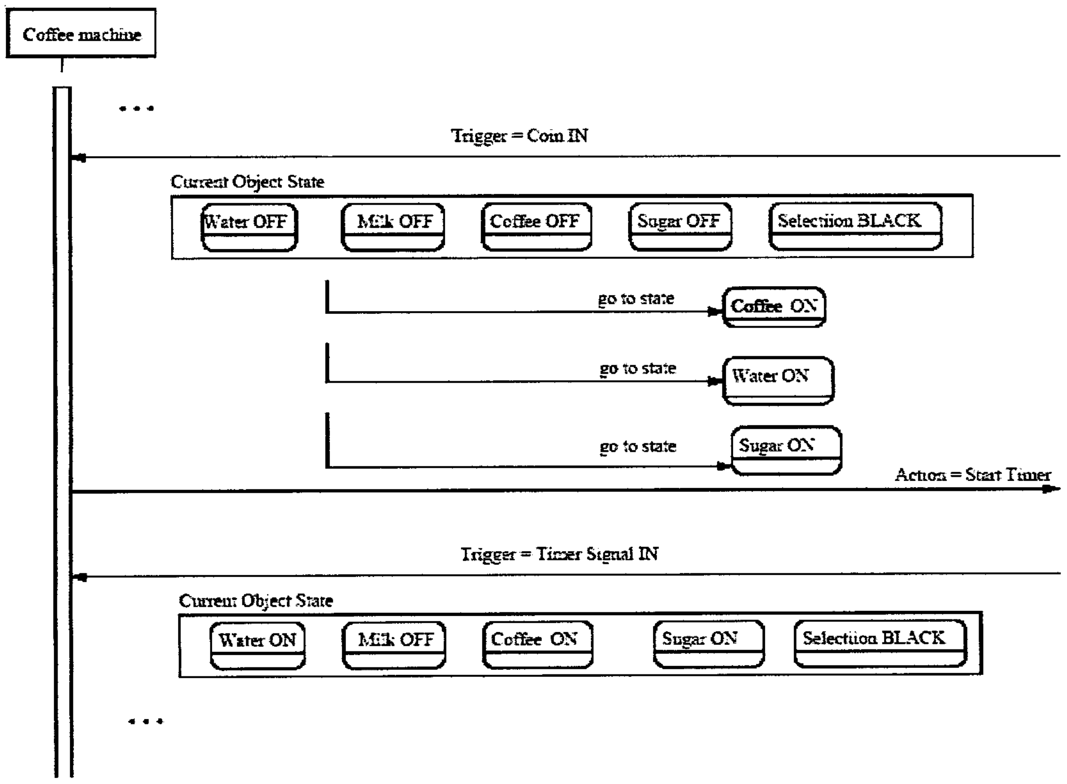

Figure 4.2 –Hybrid Sequence-State diagram fragment

Note that:
Current Object State 1 (Water OFF, Milk OFF, Coffee OFF ...) is a Persistent State
Current Object State 2 (Water ON, Milk OFF, Coffee ON...) is a Transient State

The algorithm iterates over the Sequence diagram transitions of a given object. Each Sequence transition is compared with the set of state transitions that start from the *current object state.*
In order to find whether a Sequence diagram and a State diagram of an object *obji*, which is included in the Sequence diagram, are consistent, the following steps are taken.

The Sequence diagram transitions are traversed by the algorithm. For each transition, it checks whether there is any suitable state diagram transition, i.e. a transition that has the same trigger, same actions with identical guards, same action ordering as in the Sequence diagram transition. The suitable state transitions are searched from the *current object state.* If there is a suitable state transition, the iteration continues; otherwise an error message is issued. This process of iteration and comparison is called a *run* of Sequence diagram over the State diagram. This run can be done for each object in a Sequence diagram. The State diagram that is a part of the run is the State diagram associated with the class of the object.

4.3- The algorithm

For a given object in the Sequence diagram, the algorithm iterates over the Sequence transitions. Each Sequence transition is checked for a suitable State transition, which is looked for from the *current object state*. The *current object state* is updated upon successful lookup.

The lookup starts by finding all *legs* emanating from a persistent state that are suitable for the incoming *leg* of the Sequence diagram. If the leg comparison succeeds, the algorithm looks for a suitable state *transition* that starts with this edge. Following is a description of the lookup algorithm:

Input: a persistent state and a Sequence transition being checked.

Output: a suitable State transition or a failure status.

We chose to implement a step in a hierarchical way. State machine top (means to top), which is the uppermost state in every UML State diagram, is the first state in a run. For each composite (or concurrent composite) state we must know its current configuration, i.e., its *active (sub) states* – the inner states in which the thread of control is currently located. **Each state in the current configuration might itself be a composite state and holds a configuration of its own.** Thus, a hierarchy of states is created. The conjunction of all the leaf states in this hierarchy is the /it current object state. Each composite state recursively activates the step on its active sub state *(see CompositeState::step (Sequence transition)).*

One of the main reasons we chose this method of implementation was to support State diagram *exceptions*. **An *exception is a leg emanating from some state located inside a concurrent state, to some other state outside that concurrent state.*** E.g., when an exception is received, the algorithm has to cause every thread of control inside of the concurrent state and its descendants to exit the state. Keep in mind that there might be some composite inner states inside the concurrent state. Control must leave them as well. In short, with the hierarchical approach the implementation of State diagram exception support is much easier.

Note that:

- A history state is a transient one, implemented as a state with outgoing legs targeted toward the last remembered configuration. In this way, when the history state is entered, the flow of control is directed to the last remembered inner states.
- An enclosing state of a final state is a persistent (composite) one. The execution of the Sequence diagram transition should come to an end, and the algorithm is terminated. One has to know that all the final states have been reached. A saturateable state holds a counter of the control threads that enter it. This count is updated upon each state entry. A final state also notifies its parent when a transition enters it.

```
/*- Current states include only persistent states.
CompositeState is a PersistentState. */
CompositeState::step(SequenceTransition transition) {
// suitable for concurrent composite state as well
        foreach state in the 'current object state' of this state Diagram
         PersistentState:step(SequenceTransition)
        if results have no successful entry then
                return 'none_found'
        else
                return the transition stored in the successful entry.
}
```

We will have to know to which subset of the inner states the execution has reached.

```
/* The PersistentState:step(SequenceTransition) which is single persistent state S, is
described next. */
PersistentState::step(SequenceTransition transition) {
        PersistentState::gatherResults(SequenceTransition transition)
        if any state answer encountered means that multiple options (possibilities)
        were encountered during the transition search, return it.
        count the legal results count, and if more than one,
        return multiple options.
        if no correct results were encountered, return
        PersistentState::step() for the parent state. /* No consistent states */
        else return the single correct result.
}

/* Gather Consistent states for each leg in the sequence transition */
Collection PersistentState::gatherResults(
SequenceTransition transition){
// All legs emanating from a persistent state should have a trigger,
// apart from a leg that starts a completion transition.
        for each emanating leg in SequenceTransition transition
        PersistentStateLeg::step(transition, this_state);
        collect the results (Consistent States) into a collection and return them.
}


/* seqTrans contains all legs of the Sequence diagram Transition */
PersistentStateLeg::step(SequenceTransition seqTrans,
PerstistentState S){
        compare the trigger, guard of this leg
        for conformance to the incoming leg of the seqTrans;
        3 possible actions - the exit action of the original
        state, the leg action, and the entry action are also
        checked.
        if OK then {
                return call ConsistentStateSearch(next state of this leg,
                outgoing legs of the Sequence transition).
        }
}

/* fromState is the next state of the Persistent State
OrderedCollection legs are all legs of the Sequence diagram Transition */
 PersistentStateLeg::ConsistentStateSearch(StatefromState S, OrderedCollection
legs)
{
// initially, includes only fromState parameter.
          if (state S is a JoinState) then {
               continue.
          }
          if (state S is a FinalState) then {
               continue.
          }
          // traverse legs.
          if (S is a fork state) then {
               compare all legs emanating from S with the
               upcoming legs in the OrderedCollection legs
               if (not all equal) {
                    continue.
               }
               else {
                    remove all the compared legs (L') from OrderedCollection legs.
                    insert the target state
                    of each leg L into consistent_states.
               }
          }
          else {
               foreach outgoing leg L that has no trigger {
                    compare L with the current
                    Sequence transition leg L'.
                    if L == L' then {
                         remove L' from OrderedCollection legs
                         //the function parameter.
                         insert the target state of L into consistent_states.
                    }
                    else
                         continue.
               }
          }
          if ((consistent_states include only persistent states) and
               (OrderedCollection legs are empty)) then
               return consistent_states with status 'executed'.
          else if ((OrderedCollection legs are not empty) and
               (consistent_states include only persistent states)) then
               return status 'Inconsistency'
               ./* all State Legs are different from Sequence Legs 'OrderedCollection
               legs', No consistent states  found */
          else if ((leg (L) is NULL) ) and (S was not a history state)) then
               return status 'State Diagram Finished'

}
```

5- A Run Example

5.1- Run Example A

In this Example the Algorithm Traverses the Sequence and State Diagram found on Pages 18 and 21 respectively.

```
CompositeState::step(SequenceTransition transition) {
foreach state in the 'current object state' of this state Diagram  (Wait For SellCar Trigger)

        PersistentState:step(SequenceTransition)
                (SequenceTransition = SellCar,FindBuyer,ConvinceHim)

        if results have no successful entry then
                return 'none_found'
        else
                return the transition stored in the successful entry.
}

PersistentState::step(SequenceTransition transition) {
        PersistentState::gatherResults(SequenceTransition)
        if any state answer encountered means that multiple options (possibilities)
        were encountered during the transition search, return it.
        count the legal results count, and if more than one,
        return multiple options.
        if no correct results were encountered, return
        PersistentState::step() for the parent state.
        else return the single correct result.
}
```

Leg SellCar()

```
Collection PersistentState::gatherResults(SequenceTransition SeqTrans){
        for each emanating leg in SeqTrans (SellCar)
                PersistentStateLeg::step(SeqTras, WaitForSellCar);
                collect the results (Consistent States) into a collection and return them.
        Lext Leg
}

PersistentStateLeg::step(SequenceTransition seqTrans,
PerstistentState S){
        compare the trigger, guard of this leg
        for conformance to the incoming leg of the seqTrans;
        3 possible actions - the exit action of the original
        state, the leg action, and the entry action are also
        checked.
        if OK then {
                return call ConsistentStateSearch(CheckCars , ConvinceHim()).
        }
}

PersistentStateLeg::ConsistentStateSearch (StateFromState S , OutgoingLegs
OrderedCollectionLegs )
```

```
{
        Let L be a Leg emenating from S  , L' be a Leg in OrderedCollectionLegs
                (L = ConvinceHim() , L'=ConvinceHim())

            if (state S is a JoinState) then {
                    continue.
            }
            if (state S is a FinalState) then {
                    continue.
            }

            if (S is a fork state) then {
            compare all legs L with the Legs L'
                    if (not all equal) {
                            continue.
                    }
                    else {
                    remove all the compared legs (L') from OrderedCollection legs
                    and insert the target state of each leg L into consistent_states.
                    }
            }
            else {
            ForEach outgoing leg L that has no trigger  (L = ConvinceHim() )
                    {
                            compare L with the current leg L'.
                            if  L == L' then {  (TRUE)
                            remove L' from OrderedCollection legs and insert the
                            target state of L (NextState) into consistent_states.
                            }
                            else
                                    continue.
                    }
                    Next L
            }
            if ((consistent_states include only persistent states) and (True:NextState)
                    (OrderedCollection legs are empty)) then (TRUE)
                    return consistent_states with status 'executed'.
            else if ((OrderedCollection legs are not empty) and
                    (consistent_states include only persistent states)) then
                    return status  'Inconsistency'
            else if ((leg (L) is NULL) ) and (S was not a history state)) then
                    return status 'State Diagram Finished'
}
```

Leg FindBuyer()

```
Collection PersistentState::gatherResults(SequenceTransition SeqTrans){
        for each emanating leg in SeqTrans  (FindBuyer)
                PersistentStateLeg::step(SeqTras, WaitForSellCar);
                collect the results (Consistent States) into a collection and return them.
        Lext Leg
}

PersistentStateLeg::step(SequenceTransition seqTrans,
PerstistentState S){
        compare the trigger, guard of this leg
        for conformance to the incoming leg of the seqTrans;
        3 possible actions - the exit action of the original
        state, the leg action, and the entry action are also
        checked.
        if OK then {
                return call ConsistentStateSearch(CheckCars , ConvinceHim()).
        }
}

PersistentStateLeg::ConsistentStateSearch (StateFromState S , OutgoingLegs
OrderedCollectionLegs )
{
        Let L be a Leg emenating from S  , L' be a Leg in OrderedCollectionLegs
                        (L = ConvinceHim() , L'=ConvinceHim())
                if (state S is a JoinState) then {
                        continue.
                }
                if (state S is a FinalState) then {
                        continue.
                }

                if (S is a fork state) then {
                compare all legs L with the Legs L'
                        if (not all equal) {
                                continue.
                        }
                        else {
                        remove all the compared legs (L') from OrderedCollection legs
                        and insert the target state of each leg L into consistent_states.
                        }
                }
                else {
                ForEach outgoing leg L that has no trigger  (L = ConvinceHim() )
                        {
                                compare L with the current leg L'.
                                if  L == L' then {  (TRUE)
                                remove L' from OrderedCollection legs and insert the
                                target state of L (NextState) into consistent_states.
                                }
                        else
                                continue.
                    }
                    Next L

                }
                if ((consistent_states include only persistent states) and (True:NextState)
                        (OrderedCollection legs are empty)) then (TRUE)
                        return consistent_states with status 'executed'.
                else if ((OrderedCollection legs are not empty) and
                        (consistent_states include only persistent states)) then
                        return status 'Inconsistency'
                else if ((leg (L) is NULL) ) and (S was not a history state)) then
                        return status 'State Diagram Finished'

}
```

Leg ConvinceHim()

```
Collection PersistentState::gatherResults(SequenceTransition SeqTrans){
        for each emanating leg in SeqTrans (ConvinceHim)
                PersistentStateLeg::step(SeqTras, WaitForSellCar);
                collect the results (Consistent States) into a collection and return them.
        Lext Leg
}

PersistentStateLeg::step(SequenceTransition seqTrans,
PerstistentState S){
        compare the trigger, guard of this leg
        for conformance to the incoming leg of the seqTrans;
        3 possible actions - the exit action of the original
        state, the leg action, and the entry action are also
        checked.
        if OK then {
                return call ConsistentStateSearch(NextState , ConvinceHim()).
        }
}

PersistentStateLeg::ConsistentStateSearch (StateFromState S , OutgoingLegs
OrderedCollectionLegs )
{
        Let L be a Leg emenating from S , L' be a Leg in OrderedCollectionLegs
                        (L = NULL , L'=ConvinceHim())
                if (state S is a JoinState) then {
                        continue.
                }
                if (state S is a FinalState) then { (TRUE)
                        continue.
                }

                if (S is a fork state) then {
                compare all legs L with the Legs L'
                        if (not all equal) {
```

```
                continue.
            }
            else {
            remove all the compared legs (**L'**) from OrderedCollection legs
            and insert the target state of each leg **L** into consistent_states.
            }
    }
    else {
    *ForEach outgoing leg* **L** *that has no trigger*
            {
                *compare* **L** *with the current leg* **L'**.
                if  L == L' then {
                remove **L'** from OrderedCollection legs and insert the
                target state of **L** into consistent_states.
                }
                else
                        continue.
            }
            Next L
    }
    if ((consistent_states include only persistent states) and
            (*OrderedCollection legs* are empty)) then
            return consistent_states with status 'executed'.
    else if ((*OrderedCollection legs* are not empty) and
            (consistent_states include only persistent states)) then
            return status 'Inconsistency'
    else if ((leg (L) is NULL) ) and (S was not a history state)) then *(TRUE)*
            return status 'State Diagram Finished'
}
```

5.2- Run Example B

In this Example the Algorithm Traverses the Sequence and State Diagram found on Pages 15 and 16 respectively.

```
CompositeState::step(SequenceTransition transition) {
foreach state in the 'current object state' of this state Diagram  (AnswerWait)

        PersistentState:step(SequenceTransition)
                (SequenceTransition = Answer,disconnect,connect)

        if results have no successful entry then
                return 'none_found'
        else
                return the transition stored in the successful entry.
}

PersistentState::step(SequenceTransition transition) {
        PersistentState::gatherResults(SequenceTransition)
        if any state answer encountered means that multiple options (possibilities)
        were encountered during the transition search, return it.
        count the legal results count, and if more than one,
        return multiple options.
        if no correct results were encountered, return
        PersistentState::step() for the parent state.
        else return the single correct result.
}
```

<u>Leg Answer()</u>

```
Collection PersistentState::gatherResults(SequenceTransition SeqTrans){
        for each emanating leg in SeqTrans (Answer)
                PersistentStateLeg::step(SeqTras, OnlineWait);
                collect the results (Consistent States) into a collection and return them.
        Lext Leg
}

PersistentStateLeg::step(SequenceTransition seqTrans,
PerstistentState S){
        compare the trigger, guard of this leg
        for conformance to the incoming leg of the seqTrans;
        3 possible actions - the exit action of the original
        state, the leg action, and the entry action are also
        checked.
        if OK then {
                return call ConsistentStateSearch(OnlineWait,
                (Disconnect/Answer/Connect)).
        }
}
```

```
PersistentStateLeg::ConsistentStateSearch (StateFromState S , OutgoingLegs
OrderedCollectionLegs )
{
        Let L be a Leg emenating from S  , L' be a Leg in OrderedCollectionLegs
                (L =Send/Disconnect, L'= Disconnect/Answer/Connect)
            if (state S is a JoinState) then {
                continue.
            }
            if (state S is a FinalState) then {
                continue.
            }

            if (S is a fork state) then {
            compare all legs L with the Legs L'
                if (not all equal) {
                    continue.
                }
                else {
                remove all the compared legs (L') from OrderedCollection legs
                and insert the target state of each leg L into consistent_states.
                }
            }
            else {
            ForEach outgoing leg L that has no trigger (L =Send)
                {
                    compare L with the current leg L'.
                    if  L == L' then {  (FALSE)
                    Leg Send  # Leg Disconnect
                    Leg Send  # Leg Answer
                    Leg Send  # Leg Connect
                    remove L' from OrderedCollection legs and insert the
                    target state of L into consistent_states.
                    }
                    else
                        continue.
                }
                Next L
            ForEach outgoing leg L that has no trigger (L =Disconnect)
                {
                    compare L with the current leg L'.
                    if  L == L' then {  (TRUE)
                    Leg Disconnect = Leg Disconnect
                    remove L' from OrderedCollection legs and insert the
                    target state of L (Idle) into consistent_states.
                    }
                    else
                    continue.
          }
          Next L
      }

      if ((consistent_states include only persistent states) and
          (OrderedCollection legs are empty)) then
          return consistent_states with status 'executed'.
      else if ((OrderedCollection legs are not empty) and  (TRUE)
          (consistent_states include only persistent states)) then
          return status 'Inconsistency'
      else if ((leg (L) is NULL) ) and (S was not a history state)) then
          return status 'State Diagram Finished'

}
```

6- Run Results

Run example A:

Run example A Contains in its Sequence Diagram Sequence Transition witch has 3 Legs (*SellCar, FindBuyer and ConvinceHim*). The Algorithm Iterates the State Diagram for each Leg in the Sequence Transition.

- The Algorithm starts with the Leg *SellCar()*.
The Next State of *SellCar()* is *CheckCars*, a Transient State.
The outgoing Leg from *CheckCars* State that has no Trigger is *ConvinceHim()*.

Since *CheckCars* is neither a JoinState nor a Final nor a Fork state,
The Algorithm compares each Outgoing Leg from *CheckCars* that has no Trigger, **L** = *ConvinceHim()* in this case, with the Outgoing Legs from the Sequence Transition (*OrderedCollectionLegs*), **L'** = *ConvinceHim()* Leg in this case.

The Algorithm finds that the two Legs (**L** & **L'**) are equal.

Therefore **L'** is removed from *OrderedCollectionLegs* and the Target State of **L**, *NextState* in this case, is inserted to *Consistent_States*.

At last the Algorithm Checks if Consistent_*States* include only Persistent States and if *OrderedCollectionLegs* are Empty. It finds that this is true and returns a Value 'Executed' (Validation found).

- The Algorithm continues for the second Leg of the Sequence Transition *FindBuyer ()*.

The Next State of *FindBuyer()* is *CheckCars*, a Transient State.
The outgoing Leg from *CheckCars* State that has no Trigger is *ConvinceHim()*.

Since *CheckCars* is neither a JoinState nor a Final nor a Fork state,
The Algorithm compares each Outgoing Leg from *CheckCars* that has no Trigger, **L** = *ConvinceHim()* in this case, with the Outgoing Legs from the Sequence Transition (*OrderedCollectionLegs*), **L'** = *ConvinceHim()* Leg in this case.

The Algorithm finds that the two Legs (**L** & **L'**) are equal.

Therefore **L'** is removed from *OrderedCollectionLegs* and the Target State of **L**, *NextState* in this case, is inserted to *Consistent_States*.

At last the Algorithm Checks if Consistent_*States* include only Persistent States and if *OrderedCollectionLegs* are Empty. It finds that this is true and returns a Value 'Executed' (Validation found).
- The Algorithm continues for the Last Leg of the Sequence Transition *ConvinceHim ()*.

The Next State of *ConvinceHim()* is S = *NextState*, a Persistent State.
The outgoing Leg from *NextState* State that has no Trigger is nothing.

L = Null
Since *NextState* is a Final state,
The Algorithm Continues

At last the Algorithm Checks if **L** is Null and *S* is not a History Sate (is a Persistent State). It finds that this is true and returns 'State Diagram Finished'.

Run example B:

Contains in its Sequence Diagram Sequence Transition witch has 3 Legs (*Answer, Disconnect and Connect*). The Algorithm Iterates the State Diagram for each Leg in the Sequence Transition.

- The Algorithm starts with the Leg *Answer()*.

The Next State of *Answer()* is *OnlineWait* a Transient State.
The outgoing Leg from *OnlineWait* State that has no Trigger is *Send() and Disconnect()*.

Since *OnlineWait* is neither a JoinState nor a Final nor a Fork state,
The Algorithm compares each Outgoing Legs from *OnlineWait* that has no Trigger, *ConvinceHim()* and *Send* () in this case, with the Outgoing Legs from the Sequence Transition (*OrderedCollectionLegs*), *Answer, Disconnect and Connect* Legs in this case.

The Algorithm finds that the Leg Send # Leg Disconnect and Leg Send # Leg Answer and Leg Send # Leg Connect.

The Algorithm continues for the Second Leg Disconnect and Finds it equal to Leg Disconnect in the *OrderedCollectionLegs*.

Therefore Disconnect() is removed from *OrderedCollectionLegs* and the Target State of Disconnect(), $S = Idle$ in this case, is inserted to *Consistent_States*.

At last the Algorithm Checks *OrderedCollectionLegs* if they are not empty. It finds it true since Answer() and Connect() still in *OrderedCollectionLegs,* and consistent_states include only persistent states, true since it includes Idle State, so returns 'Inconsistency'.

7- Conclusions

7.1- Partial legs ordering in a Sequence diagram

Consider running a Sequence transition. We might not care about the order of some of the outgoing legs. We thought of some expansion of UML where one can actually visually set **partial ordering** of the Sequence diagram legs. Legs among which the order is not important can emanate from the same spot in the Sequence diagram. This complicates greatly the running algorithm, but note that it still remains Consistent. This is not implemented in the project.

7.2- Compound initial transition

The UML standard is somewhat vague about the transition concept, as we mentioned earlier. We assumed that there is a single empty transition from an initial state to a persistent state. However, UML also uses the term compound transition. We might have a transition emanating from the initial state, which includes several choice states for example (which will make it several transitions, not the standard). Moreover, it might even end outside the composite state. Note that in this case it's much harder to locate the flow of control, because it's cumbersome to make hierarchical running algorithm like this.
One might consider global current states instead of current states per every composite state, in this case. This makes exception control harder and time-costly, though one will have to detect exception, and then undo the steps of the states that were in the same father of the exception receiving step. Note also that the exception can go up several levels in the hierarchy. Then the steps taken in these levels will have to be undone. This makes exception much harder to implement.

7.3- Summary

Our Algorithm provides a simple solution for the UML dynamic diagrams consistency problem. It is more than a Boolean result consistency checker; it is a tool that enables fast error detection and recognition.

Its main advantage is that it is straightforward to use and the output is easy to understand, unlike some other similar tools, and the input is given in a straightforward form of a Sequence diagram. No first or second order logic knowledge is needed. Such knowledge is not always present at the client side (analysts and designers).

Its main disadvantage is that it gives one information about *specific* runs that are designed in a Sequence diagram, and not any possible runs. The solution of this would probably require solid computer science logic background at the client side.

8- References

[1] Software Engineering 6th Edition by IAN SOMMERVILLE

[2] Object Management Group. OMG unified modeling language specification. Found at *http://www.omg.org/cgi-bin/doc?formal/00-03-01*. And *http://www.omg.org/cgi-bin/doc?formal/01-09-67*.

[3] *http://www.iioss.org/About/index-e.html*

[4] The IIOSS project. Integrated Inter-Exchangeable Object-Modeling and Simulation system. *http://www.iioss.org/index-e.html*

[5] *http://www.objectmentor.com/courses/resources/articles*

[6] Object Oriented Analysis and Design Principles *www.selectbs.com*

[7] Center for Software Engineering *www.sunset.usc.edu/cse*

[8] The MOSS project. Integrated Inter-Exchangeable Object-Modeling and Simulation system.

http://www.iioss.org/index-e.html.

[9] R. Martin. UML tutorial. http://www.objectmentor.com/courses/resources/articles/umlsequencediagrams.pdf. Cf. Race conditions chapter.