Innovations Syst Softw Eng (2011) 7:1–8
DOI 10.1007/s11334-010-0138-8

REVIEWS

An approach for modeling a formal Use Case Type at early development phase without loosing abstraction

Francisco Supino Marcondes · Ítalo Santiago Vega · Luiz Alberto Vieira Dias

Received: 7 July 2008 / Accepted: 1 October 2010 / Published online: 21 October 2010

Abstract To model formal Use Case Type is useful in using a state machine diagram once it is verifiable through mathematical analysis, and it is very precise. But by the same characteristics, it is very difficult to be modeled in early phases since many definitions are still open and vague. This study presents an approach to obtain a formal Use Case Type definition in early phases without losing the intuitive events searches provided by other heuristic models, usually sequence diagrams.

Keywords Use Case Type · Model transformation · State machine · Formal Use Case

1 Introduction

To model Use Cases, the modeler should describe it in terms of a dialog between actors and the system. The dialog is defined based on stakeholder's needs encapsulating the system's control in a way which are just defined in-coming and out-coming events, which are often modeled by sequence diagrams [1].

This diagram models the sequence of events, considering the order of its occurrence, and representing the interaction of many actors (if it is used to model Use Cases) with the system. Considering the Use Case Type [2], those diagrams are not appropriate, since this kind of model many times do not represent a sequential behavior, for the actors can asynchronously trigger events. In order to complete the whole description of the Use Case and refine its Type, it is usual to make 'n' scenarios, searching for alternative flows which were not yet modeled.

That approach causes a big waste of time to elaborate many possible situations, for the Use Case Type model refinement. This is necessary since this diagram shall represent the whole life cycle, so it is needed to consider many situations in order to get the overall behavior of the Use Case, searching exhaustively for a non-considered situation.

Even considering lots of situations, the modeler never get a totally complete life cycle, which will consider all possible sequences to events of the Use Case as expected to describe its Type. As stated, the sequence diagram should follow a sequence to represent a system's behavior, but the triggers are not sequential, so, to 'n' steps are $n!$ possible situations of the trigger sequence permutations. To illustrate the impossibility of getting the Use Case Type by Sequence diagram, suppose a 8-message diagram, it will produce $8! = 40{,}320$ different situations. In fact, in this example, each event is supposed to occur just once, which is not true in a real situation because, if there is permutations with repetitions, the complexity of the model increases, thus making it very difficult to obtain all situations, and the number of scenarios can be infinite. So, to those situations, the modeler can just make some scenarios to test the flow.

Despite the impossibility of model, a proper Use Case Type using sequence diagrams, it has another problem, as the diagram grows and became complex, it starts to hide inconsistencies, contradictions, and ambiguities troubling the analysis procedure.

F. S. Marcondes (✉) · L. A. V. Dias
Brazilian Aeronautics Institute of Technology,
Praça Marechal Eduardo Gomes, 50-Vila das Acácias,
São José dos Campos, SP, Brazil
e-mail: yehaain@gmail.com

L. A. V. Dias
e-mail: vdias@ita.br

Í. S. Vega
Pontificia Universidade Católica de São Paulo,
Rua Monte Alegre, 984-Perdizes, São Paulo, SP, Brazil
e-mail: italo@pucsp.br

So, it should be a way that allows the proper model of the Use Case Type, which represents all '*n*' situations without the ambiguity, inconsistency and contradictions problems in order to improve the requirement engineering, turning it into an agile, detailed and less complex procedure.

Those objectives can be achieved by the use of the state machine diagram to model the Use Case Type, and it includes the benefit of being a formal, mathematical-based model, which includes the Mealy and Moore machines algorithms with more resources to improve its manipulation. The only problem of using it is that at the early phases of a software development, it is difficult to identify events and states to the state machine. This occurs in the early stages of the development, since the systems are not well defined yet, and the aspects needed by state machines can be so abstract that these complicates the process rather than better. In other hands to get diagrams like activity or sequence diagrams are easier to collect, but retrieve to the problem stated before.

The purpose of this research is to present an approach that allows the modeling of Use Case Type, using State Machines diagrams, without loosing the intuitive aspect that can be modeled in early phases of the development in order to decrease the difficulty in obtaining states and triggers, without the sequence diagram problem stated before.

2 About models

The UML's official guide defines the term Use Case as the same concept as for the Use Case Type, and it can be assigned to one or more diagrams, which represent its behavior [2]. An instance of a Use Case, also called scenario, refers to one logical path of the Use Case Type behavior [3].

On this way, to model the Use Case Type behavior is a hard task once it has to predict all possible situations. The other option is to try to obtain the all possible scenarios of the Use Case with flowcharts. It is easier to model flow of events with the sequence, or the activity diagrams than to utilize the state machine diagram [4].

Even for small software systems, those scenarios can be so numerous that is not possible to represent them all. It leads the analyst to choose some "most important" scenarios that will be modeled, resulting in unpredictable behavior of the system [5]. Some of those unpredictable situations will be discovered, analyzed, and treated at the test phase incurring in increased costs, and in some cases, getting the project under risk [5].

Besides, the first choice, using state diagrams, is the safer one, for to model state machines is not a trivial task. States can be defined as a condition of existence of an instance that is distinguishable from other one [6] or as a situation in which some condition is held [2]. Turing [7] called state as a condition, and the pair (condition, scanned symbol) as a configuration [7]. Abstracting Turing's definition to the UML's State Machines, state is a set of properties whose values determine how the system will respond to an event [3], and a transition (configuration) is the pair (state, event), which determines the future state (condition) of the system.

It is difficult to obtain those properties, and identify the states in early phases of the development, for at the Use Case description phase there are not yet entities or variables, so the states are often modeled to an appropriate state machine using the modeler imagination. In other hand, scenarios, specially the basic ones, can be easily modeled if the modeler could be helped by the stakeholder. This study presents an approach on how to obtain the Use Case Type behavior starting from a scenario description.

The UML's behavior state machine is a specialization of the Harel's state machines for objects [2], which was created by the union of the Moore and Mealy machines [8]. This statement is important to a proper understanding of the model and transformations of State Machines, since they inherit all Mealy/Moore algorithms and properties.

Another advantage of that statement is the finite state machine definition used by Mealy/Moore machines, which is the best known representation of the von Neumann architecture. It has input, output and a well-defined flow representing the control, since those machines only recognize regular expressions so, there is no memory need. An extension to the Harel's model can include a stack or tape to increase the computational power to represent the von Neumann's architecture memory. Thus, the state machine model is a powerful tool to describe a system's behavior.

Sequence diagrams were originally defined to show interactions between objects in the sequential order that those interactions occurs [2]. On essence, it can model interactions between anything in a simple way [2], and it was usual to model Use Case scenarios such as an interaction between an actor and a system [4]. To model this diagram is as easier as to model the activity diagram, it is usual that the stakeholder knows what is needed to introduce, and what is expected back of the system, at least on the expected basic flow.

From a requirement viewpoint, the activity diagrams should be modeled if it will help to find events. Use Cases are focused in events and its principal characteristic is encapsulation, so, control rules should be avoided.

Interactions could be synchronous or asynchronous, and are characterized by a message sending process that results from actions, which may be actions due by other behaviors [2]. In fact, messages works as message events [2], an event can be defined as the specification of some occurrence that may potentially trigger effects by an object [2]. Events could be (1) call, (2) message, (3) signal, and (4) time [2]. Those events trigger actions at the object that receives the event. Trigger relates an event to a behavior that may affect an instance of the classifier [2].

As sequence diagrams are powerful to model interactions, it is poor to model conditions, as well, activities diagrams are poor to model interaction, even using swim lanes, once it was not originally created do model those situations. To model activity diagrams is necessary to identify the actions performed by the system triggered by events, which need to be modeled by the sequence diagram.

Neither the sequence diagram support activities, neither the activity diagram support events, but both are fundamental in describing the system's behavior over many possible scenarios. Even those models producing similar diagrams represent different views as stated before, so the modeler should produce both diagrams to an appropriated knowledge of the Use Case, but in this case, the modeler will produce a prohibitive number of diagrams multiplied by two.

The only diagram that supports both events and activities is the state machine [2]. So, in thesis, it is possible to obtain a state machine from the sequence diagrams, without need to identify the variables. If this is a true statement, by an initial state machine, it is possible to make it deterministic in order to represent the overall scenarios of the Use Case obtaining a formal definition of the Use Case Type.

3 The approach

State machine diagrams have two important properties: one is responsible to trigger the transitions and are modeled as events, while the other is responsible to perform actions. As stated before, sequence diagrams are useful to identify events, which can be mapped to a state machine diagram as each event generates a transition to one new state [9].

3.1 Equivalence over sequence diagrams and state machines

Hypothesis (1) Each object into a sequence diagram has an equivalent state machine which represents its behavior.

Base of hypothesis (1) On sequence diagram, one general state is inserted before and after each event, as show at Fig. 1.

Actually the $e2$ is a special kind of event onto sequence diagram because it represents a return statement [9], it causes execution to leave the current subroutine, and resume at the point in the code after where the subroutine was called [10]. From a practical viewpoint, the return statement is considered just as a common event. Since it is an existing object, it has an state before the event and after the return statement if it was not created or destroyed by the event. On those situations, it can be created a λ state before, in case of creation, or after, in case of the object destruction, which will be eliminated at the end of the process, or based on Harel's Statecharts, there is a transition from a pseudo start state to the first state and a transition from a last state to a pseudo final state, thus both cases can use this notation.

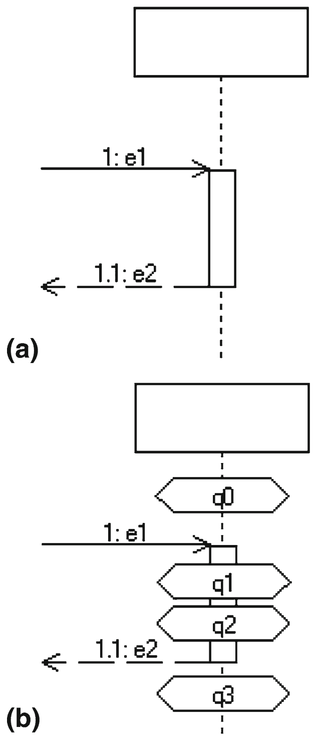

Fig. 1 **a** A generic sequence diagram. **b** A generic sequence diagram with its states

Considering two events $e1$ and $e2$ of a sequence diagram, the approach produces two independents automata M, which have the transition $(q0, e1 \rightarrow q1)$, Fig. 2a, and N which has the transition $(q2, e2 \rightarrow q3)$, Fig. 2b. Since they are regular expressions, they can be concatenated producing a machine K = M · N [11] having the $(q0, e1 \rightarrow q1), (q1, \lambda \rightarrow q2), (q2, e2 \rightarrow q3)$ transitions, Fig. 2c. In general terms, the expression will be: $(genericState0 : State, event : ofSequenceDiagram \rightarrow genericState1 : State)$ $(genericState1 : State, \rightarrow genericState2 : State)$ $(genericState2 : State, event : Sequence Diagram \rightarrow genericState3 : State)$.

Over the K machine, it can be applied a transformation that eliminates the λ transition [12] producing the nondeterministic finite state (NFS) machine K $(q0, e1 \rightarrow \{q1q2\})$, ({q1q2}, e2 → q3) as shown at Fig. 3. In this text, only the λ-free structure will be used to simplify the explanation. In general terms, $(q0, event : ofSequenceDiagram \rightarrow q1)$ $(q1, event : SequenceDiagram \rightarrow q2)$.

a. Induction of hypothesis (1) to n events On sequence diagram, one general state is inserted before and after each event. Considering each event of a sequence diagram, the approach produces 'n' independents automata since 'n' is equal as the number of events. Each machine has the transition (a precedentState, event → a posteriorState). Since they are regular expressions they can be concatenated producing a machine K = machine 'n' · machine 'n'+1· . . . ·

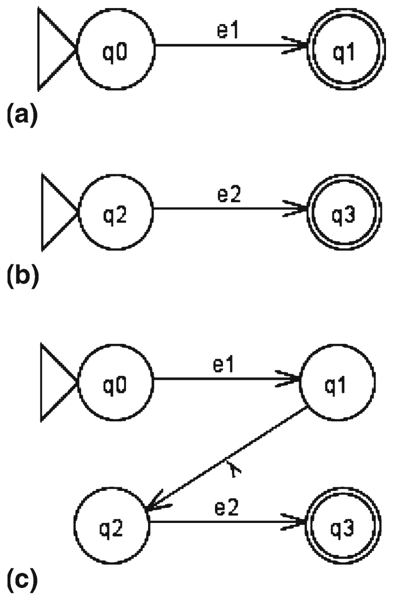

Fig. 2 **a** M machine, **b** N machine, **c** K machine

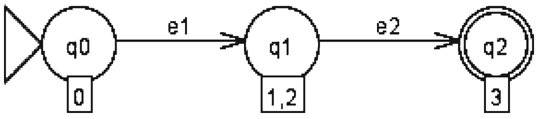

Fig. 3 NFS K machine without λ transition

machine ‘n’+($n-1$) having the (initial state of machine ‘n’, event of machine ‘n’ → final state of machine ‘n’), (final state of machine ‘n’, λ → initial state of machine ‘n’+1), (initial state of machine ‘n’+1, event of machine ‘n’ → final state of machine ‘n’+1) . . . (final state of machine ‘n’+($n-2$), λ → initial state of machine ‘n’+($n-1$)), (initial state of machine ‘n’+($n-1$), event of machine ‘n’ → final state of machine ‘n’+($n-1$)). Over the K machine, it can be applied a transformation that eliminates the λ transition [12] producing the NFS machine K.

This machine represents with great fidelity the behavior of the object described at the diagram, so, as stated “each object into a sequence diagram has an equivalent state machine, which represents its behavior”.

b. Induction of hypothesis (1) to n objects To ‘n’ objects, obtain the state machine to each one. Since each object is independent and state machine intend to model objects behavior, each machine is disjoint or orthogonal to the others, so as stated at the hypothesis, “each object into a Sequence diagram has an equivalent state machine, which represents its behavior”.

c. Induction of hypothesis (1) to n diagrams One same state machine can represent ‘n’ possible path of execution of an object, but the Sequence diagram can represent one at a time. To ‘n’ diagrams of one same object, obtain the equivalent state machine, use the union operation in the generated machines. On the obtained machine, apply the Myhill–Nerode minimization to obtain the final machine.

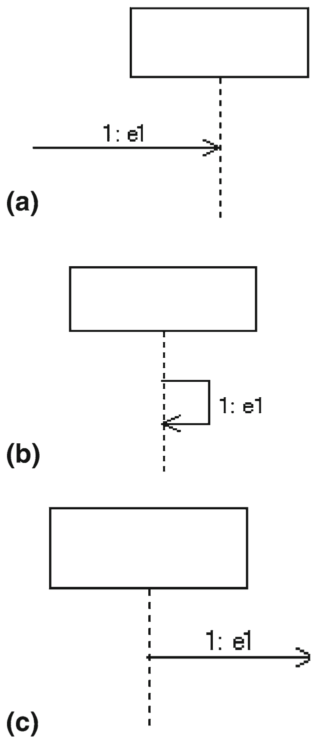

Fig. 4 Possible executions of $e1$

Hypothesis (2) Each state machine that represents an object behavior has ‘n’ equivalent sequence diagrams that describes it.

Refutation of (2) Suppose a K Machine with the transition ($q0$, $e1 \rightarrow q1$). Since a state machine represents the behavior of an object, there is one object which implements that machine. Each trigger of the state machine is triggered by an event, so it can be considered as event occurrence or just an event, so, for each trigger of the state machine there is an equivalent event.

To create the sequence diagram, choose some desired path of execution in the state machine to define the order of event occurrence at the sequence diagram. Create the sequence diagram and insert events on the defined order. At this time, it is impossible, even using the correct definition of events (call, message, signal, and time), to decide where the event is started as shown at Fig. 4. That occurrence is based on the nature of the state machine, which is defining a behavior for one object in a regular-context way.

3.2 Approach to transform sequence diagram into a complete FSM

Approach From a sequence diagram that represents the basic flow of a Use Case, obtain the equivalent state machine E1.

The result of this procedure is a non-deterministic Mealy machine, in which each transition has its own states and actions. On this machine, it can be applied an algorithm to transform it into a deterministic machine, this transformation has to be made manually, because it involves many project decisions as shown in the case study.

Once obtained the deterministic state machine diagram, the modeler can give names to the general purpose states finishing the Use Case Type modeling.

3.3 State machine complexity

Hypothesis of Lemma (1) At a state machine, an increase the number of states implies in less model complexity than an increase in the number of events.

Base of Lemma (1) Suppose a state machine with four states and three transitions, it yield $4^3 = 64$ possible situations to the system execution. Suppose a state machine with three states and four transitions, it results in $3^4 = 81$ possible situations to the system.

Induction of Lemma (1) Suppose a state machine with x states and y transitions, it yields x^y possible situations to the system execution. It can be reduced to the elementary true, that in most cases, $x + 1^y < x^{y+1}$. So, as stated at a state machine, an increase in the number of states implies in less model complexity than an increase in the number of events.

4 Case study

This approach was applied to model a simplified attitude and orbit control system (AOCS). The Use Case modeled was called as simulate satellite dynamics, and it performs the integration of dynamics equations with the board computer every 5 ms by some parameter defined by the researcher. The Fig. 5 shows the sequence diagram that represents one situation of the Use Case Type; it is a simple model since there were not yet considered its scenarios nor its alternative situations.

In some projects, it is useful to use an activity diagram to represent the initial expectation of the stakeholder from where some events can be identified, in this study this kind of activity was omitted since it is out of scope.

Figure 5 has both synchronous and asynchronous events, so the modeler to get the complete Use Case Type needs to insert the asynchronous events in many places. An important example is the stop event, the researcher can stop the simulation everywhere, to proper represent it. The modeler should produce six models inserting the stop event after each event. As well, the clock will send the wake up command when it realizes that was already passed 5 ms, the signal can come

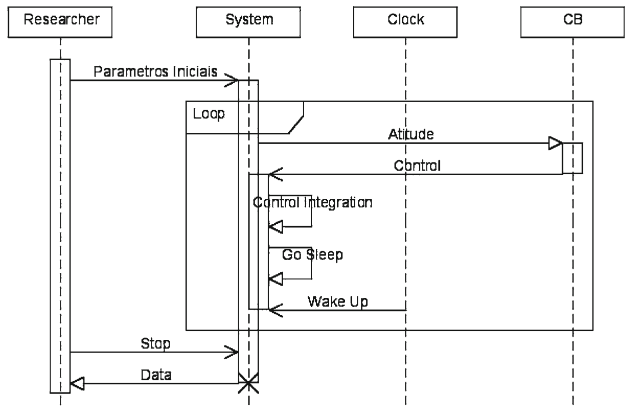

Fig. 5 Simplified Use Case Type

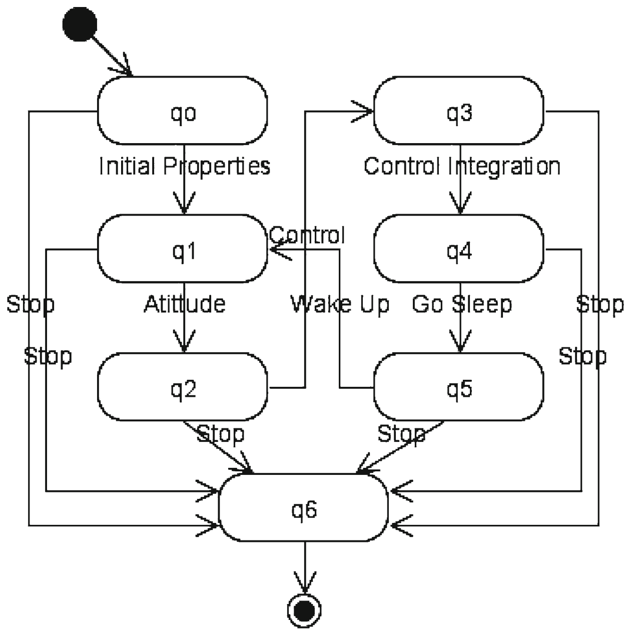

Fig. 6 State machine generated by the Use Case sequence diagram

wherever state is the system, so it can be very complex to model those situations using sequence diagrams.

The approach was applied over Fig. 5 producing the model as appear at Fig. 6. Figure 6 already considers the stop condition to all places at the system, so, where it was needed +6 diagrams. On the other hand, by the approach used just one, and the system can grow as far as it needs, but it will be still modeled by just one diagram.

Considering that state machines are based on Mealy and Moore machines, a non-deterministic machine can become a deterministic one, this can be done manually or by existing algorithms. The main idea of this transformation is to obtain the overall situations of the Use Case Type without retrieving to ambiguity, contradiction and inconsistency problems. State machines used to avoid those problems. After this transformation, the machine can became too complex (still less complex than the Sequence diagram) to be modeled as a diagram, so it is a better approach to use a table to represents it.

In Table 1, were placed the basic transitions, and manually were defined some alternative ones, this activity leads to some project decisions which on the scenario-based approach are often missed, but in a state machine just blow-up. To the case study, the system's behavior decisions were blown up:

1. The researcher can modify the properties of the satellite during the simulation?
   Actually this kind of question occurs to the user when he wants to do this action and realizes the system do not consider that possibility, so in the worst possible phase, where the costs are 1,000× greater than if it was identified at the early phases. Using this approach, since the state machine requires for model its entire life cycle, this modeler's decision just blown up. By this example, the researcher can change the satellite configuration at any state, if that event occurs the system shall send the new attitude to the board computer, so, it went to $q1$ from everywhere.
2. What happens if the system receives a wake-up event during $q3$ or $q4$?
   This kind of situation represents a deadline lost, which implies the creation of one new state to proper treat it. The answer of this question also yields the solution to other problems, which the modeler not yet considered, like what happens if the system receive a wake-up just after it received configurations properties, so by filling in the blanks the modeler will resolve many system properties without need to consider them all. Mature modelers can identify best patterns of events, an if treat them appropriately, they do not need to identify case by case to resolve them, he can just extrapolate to the 'blanks' resolving similar questions.
   Another advantage of using state machines is its polymorphic characteristic, so, the new state on the state machines implies that the system will act different to resolve the dead line lost until return to the original flow. This characteristic is very desirable to the real-time problems.
3. What happens if the system receives a wake-up event during when it is in the 'DeadLine lose' state?
   That flow, in this case study, can indicate a more seriously problem with the system, maybe some actor was crashed. This possibility leads to the refinement of the system; maybe it is important to verify the correct behavior of the actors, including filters or redundancies, resolving those questions the system should grow until all life cycle properties are resolved. To keep the model as simple as possible, in this study the situation leads to an error condition.

Similar decisions and refinement will blow up form the detailed analysis of the state machine. To this example, suppose the modeler resolved all decisions of the system and still have some blanks at the transition table. That implies the machine as a non-deterministic one, so, the modeler can apply the conversion to a deterministic machine using the algorithm proposed to similar purpose for Mealy and Moore Machines [11]. That procedure produces a new state to where all blanks should go, and applying the minimization as proposed by Marcondes et al. [13], the new state and the 'Error' state will be concatenated producing a new state called also as 'Error' and waits the researcher to stop the system.

The advantage of that approach causes the system never to halt since all events are well monitored and been a mathematical-based tool, there are no place for ambiguity, contradiction and inconsistency.

Table 1 shows the transition table of the case study, it has all transformations described above including the minimization.

Since this case study is modeled by the Use Case approach, this formal, deterministic and well-defined description will

Table 1 State machine transitions

	q0	q1	q2	q3	q4	q5	q6	DeadLine lose	Error
Initial properties	**q1**	q1	q1	q1	q1	q1	q6	q1	Error
Attitude	Error	**q2**	Error	Error	Error	Error	q6	q2	Error
Control	Error	Error	**q3**	Error	Error	Error	q6	q3	Error
Control Integration	Error	Error	Error	**q4**	Error	Error	q6	Error	Error
Go sleep	Error	Error	Error	Error	**q5**	Error	q6	Error	Error
Wake up	DeadLine lose	DeadLine lose	DeadLine lose	DeadLine lose	DeadLine lose	**q1**	q6	Error	Error
Stop	**q6**	**q6**	**q6**	**q6**	**q6**	**q6**	q6	q6	q6

The basic flow appears bold

Table 2 Use case applications [6]

Phase	Application
Analysis	Suggest large-scale partitioning of the domain
	Provide structuring of analysis objects
	Clarify system and object responsibilities
	Capture and clarify new features as they are added during the development
	Validate the analysis model
Design	Validate the elaboration of analysis in the presence of design objects
Coding	Clarify purpose and role of classes for coders
	Focus coding efforts
Test	Provides primary and secondary test scenarios for the system validation
Deployment	Suggest iterative prototypes for spiral development

collaborate to a best definition of the subsequent phases. Table 2 defined by Douglass et al. [6] presents the Use Case usage on the software development process.

Despite its formalism, it is an easy to understand model that represents all possible situations of the Use Case Type, at this case study, it was represented by $9^7 = 4782969$ situations. On those situations, it can be defined by scenarios that will test the system's behavior. Another advantage of the state machine usage is the automatic-like definition of the Object Constraint Language (OCL) completing the whole description of the life cycle of the Use Case [2].

Until this point of the analysis, the states received pseudo-names to be proper optimized. The caution about the optimization over pseudo-states does not lead to lose abstraction, since it is difficult to choose a proper name for optimized states if the modeler named them before, as well it is difficult to define good names in early stages of the modeling. After the optimization process made over a mature model, the modeler should give names in order to increase the semantics and readability of the model, becoming as in Table 3.

The transition table, modeled in Table 3, representing the 9^7 situations of the system, the calculus was based upon permutations with repetitions technique, and could be modeled in just one diagram. This approach produces a robust diagram once all possible situations are modeled and can be tested. The Use Case Type represents a set of requirements of the system and they can be easily derived from that state machine, analyzing each state and its transitions and actions.

5 Conclusion

The Use Case Type modeling by state machine is a hard task if it is made in early development phase by its dependency on variables and its values. The variables change is triggered by events and procedures which can be easily modeled in early phases supposing there is a set of variable values before an event and a different state of the variables after the event occurrence. The changes made by actions have the same procedure.

This study has shown the approach to transform sequence diagrams into a state machine diagram by formal transformation, and a way to obtain the Use Case Type over it. The benefit of this approach, besides the Use Case Type modeling, is the possibility of using Mealy and Moore algorithms, which can be used to optimize, transform, and making it deterministic.

This paper has used, in particular, the deterministic and minimization transformation getting robust Use Case Type behavior representing 9^7 situations of the Use Case Type in

Table 3 State machine transitions

	Waiting properties	Setted	Waiting contol	Dynamics propagation	New atittude	Sleeping	Stopping	DeadLine lose	Error
Initial properties	**Setted**	Setted	Setted	Setted	Setted	Setted	Stopping	Setted	Error
Attitude	Error	**Waiting control**	Error	Error	Error	Error	Stopping	Waiting contol	Error
Control	Error	Error	**Dynamics propagation**	Error	Error	Error	Stopping	Dynamics propagation	Error
Control integration	Error	Error	Error	**New attitude**	Error	Error	Stopping	Error	Error
Go sleep	Error	Error	Error	Error	**Sleeping**	Error	Stopping	Error	Error
Wake up	DeadLine lose	DeadLine lose	DeadLine lose	DeadLine lose	DeadLine lose	**Setted**	Stopping	Error	Error
Stop	**Stopping**	**Stopping**	**Stopping**	**Stopping**	**Stopping**	**Stopping**	Stopping	Stopping	Stopping

The basic flow appears bold

only one diagram. It is not the intent to be a perfect, defect free model, once the input can have failures, but it formally yields a correct initial diagram to the modeler work over it, as shown in Table 3.

Finally, constructing and making analysis on only one model can increase the development velocity, improve the software quality, and decrease the defect injection rate, in comparison with the same processes made over sequences diagrams.

References

1. Donald B (2004) UML's sequence diagram. In: IBM. http://www.ibm.com/developerworks/raration/library/3101.html. Accessed 20 Jun 2008
2. Object Management Institute (2007) OMG unified modeling language (OMG-UML). Superstructure, v2.1.2 OMG
3. Rational Software Corporation (2006) DEV470: rational rose real-time (student material)
4. Larman C (2004) Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development. Prentice-Hall, Englewood Cliffs
5. Dias LAV (2008) Lecture notes. Aeronautics Institute of Technology. ITA, São José dos Campos, SP, Brazil
6. Douglass BP (2006) Real time UML. Addison-Wesley, Boston
7. Turing AM (1936) On computable numbers, with an application to the Entscheidungs problem. Proc Lond Math Soc 42:230, 265 [ibid 43:544–546 (1936)]
8. Drunsinsky D, Harel D (1994) On the power of bounded concurrency I. ACM 004-5411/94/0500-0517
9. Pender T (2004) UML Bible. Wiley, Hoboken
10. Sebesta RW (1999) Concepts of programming languages. Addson Wesley, Boston
11. Hopcroft JE, Motwani R, Ullman JD (2001) Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Boston
12. Java Formal Language Application Tool (JFlap) (2007). In: JFlap. http://www.jflap.org. Accessed 24 Nov 2007
13. Marcondes FS, Colonese E, Vega IS (2008) Dias LAV proposing a formal method to reduce RTS logical model complexity. doi:10.1109/ITNG.2008.202