Journal of Universal Computer Science, vol. 19, no. 1 (2013), 53-77
submitted: 6/8/10, accepted: 28/12/12, appeared: 1/1/13 © *J.UCS*

Deriving System Behavior from UML State Machine Diagram: Applied to Missile Project

Hyun-Seok Min
(Korea University, Seoul, Korea
hsmin@formal.korea.ac.kr)

Sang-Mun Chung
(Agency for Defense Development, Daejeon, Korea
smchung@add.re.kr)

Jin-Young Choi
(Korea University, Seoul, Korea
choi@formal.korea.ac.kr)

Abstract: Traditionally, System Analysis and Software Design are treated as separate processes. Software Design is based on System Analysis but they have little direct relationship to each other. UML (Unified Modeling Language) is widely accepted by industry as the de facto standard for System Analysis and Software Design. The primary tool for System Analysis is the Use Case Diagram and its Scenarios, while the primary tools for Software Design are the Class Diagram and Sequence Diagram. State Machine Diagram is also very useful for behavioral modeling. Our aim is to derive system behavior from software design, so that separate processes can work together. This paper suggests how to make a system-wide State Machine Diagram by gathering State Machine Diagrams in the system using predefined Stereotypes and Synchronization/Externalization. The resulting system-wide State Machine Diagram can be used for various things, such as automatic test case generation for the system, deadlock detection, and Use Case scenario consistency checking. The proposed method is applied to the Missile project of ADD (Agency for Defense Development) in South Korea. The result is very promising. It is expected to be applied to more projects.

Keywords: System, UML, State Machine Diagram, Test, Deadlock, Consistency
Category: D.2.4, D.2.5, F.3.1, F.3.2, I.6.4

1 Introduction

UML is accepted as a de-facto standard modeling language for software design. UML 2.2 (the current version at the time of writing this paper) has 14 diagrams; each diagram shows a specific aspect of the desired system. Some of the diagrams (such as Class Diagram, Object Diagram, Package Diagram) show the static aspect of the system, while some diagrams (State Machine Diagram, Activity Diagram) describe dynamic aspects of the system. Other diagrams show interactions between various elements in the system (Sequence Diagram, Communication Diagram, Interaction Overview Diagram, and Use Case Diagram)

State Machine Diagram, which is popular for describing the behavior of the class, is a formally defined diagram. If a State Machine Diagram is used in design, source

codes can be generated from it. However, the State Machine Diagram needs to be used as a behavior of one class to enable the generation of source codes. Popular CASE (Computer Aided Software Engineering) tools use this approach to generate source codes from State Machine Diagram.

Therefore, if system-wide behavior is needed, a system-wide State Machine Diagram needs to be made separately. This may cause inconsistency with the individual State Machine Diagrams. If we want to perform integration testing or system testing, we need to make test cases manually, or to make a system-wide State Machine Diagram first and then generate test cases from it. A system behavior check, such as deadlock/livelock and/or property checking, is more important than ever, with increasing prevalence of multi-threaded programs. Since the design used for source code generation is usually for one class, it is hard to predict overall system behavior with individual design. These imply system-wide behavior is needed and this behavior may not conflict with each State Machine Diagram of classes.

2 Related Research

Hartmann et al. [Hartmann 00] proposed an approach to combine more than one State Machine Diagram, making a system behavior State Machine Diagram. The goal was to generate test cases for the system that is composed of several components. The components are COM/DCOM/CORBA components. They run independently and communicate with each other. Each component has a State Machine Diagram as a behavioral model. State Machine Diagrams communicate to others via events. Event communication is well explained in CSP (Communication Sequence Process) [Hoare 87] and [Hartmann 00] uses a similar concept and notation. The global State Machine Diagram is composed incrementally and reachability is computed to remove unnecessary states. Test cases are generated and executed after the global State Machine Diagram is composed. In this work, two system-wide State Machine Diagrams are merged into a global State Machine Diagram. (It is basically a graphical CSP using the same syntax.) Developers need to make system-wide State Machine Diagrams manually first and then merge them to use this approach. While using a State Machine Diagram itself to make a system behavior is uncommon, using Model Checkers to check the system behavior with State Machine Diagram has been popular. Kaveh and Emmerich have used State Machine Diagram for deadlock detection [Kaveh 01]. This work has <<synchronous>> Stereotype in State Machine Diagram transition and translates the diagram into FSP (Finite State Processes) for analysis. Mikk and Lakhnech, et al. [Mikk 98] and Lilius and Paltor [Lilius 99] use PROMELA/SPIN for the analysis. vUML [Lilius 99] uses UML Stereotype for design and automatically generates the PROMELA specification from UML design. Conceptually, the approach taken in this paper is similar to the one in these papers. However, the biggest difference is that our approach explicitly uses the external stimuli concept. Thus, it is more flexible in the system behavior selection. In addition, only State Machine Diagrams are used in this paper compared to the translation to other language, such as PROMELA/SPIN.

One of the reasons of making a system State Machine Diagram is to automatically generate test cases for the system. There is much research on generating test cases from a State Machine Diagram ([Kim 99], [Offutt 99], [Kansomkeat 03]).

Kim et al. [Kim 99] proposed the strategy introducing Extended Finite State Machines (EFSMs) by removing concurrent states and state hierarchy to generate test cases. Briand et al. [Briand 03] used OCL as an input condition in the State Machine Diagram. Gnesi et al. [Gnesi 04] shows theoretical background to generate test cases from State Machine Diagram. Prior to UML 2.0, the Activity Diagram was very similar to the State Machine Diagram. [Wang 04] proposes using an Activity Diagram to generate test cases. However, they are not system-level approaches but class-level approaches.

Some research considers system-level test case generation. The majority of work uses Use Case and its scenarios. The Use Case Diagram is ideal for system test case generation, since the Use Case Diagram shows interactions between the Actor and the System. Normally, the Use Case Diagram is accompanied by a textual description or other diagrams, such as Sequence Diagram, Activity Diagram, and State Machine Diagram, which is termed a scenario. Briand and Labiche [Briand 01] proposed an approach using the Use Case Diagram and its accompanying Activity Diagram to generate system-level test cases. In this work, the Activity Diagram is used as a hybrid of the Sequence Diagram (showing message interaction between instances) and State Machine Diagram (having many alternative paths). Combining two or more diagrams to generate system-level test cases is a common approach. Nebut et al. [Nebut 06] used the Use Case Diagram and its accompanying Sequence Diagram to generate test cases. One approach taken in this work is the use of the pre/post-conditions of Use Cases to automatically generate input data and test oracle. Froehlich and J. Link [Frohlich 00] proposed a method to make a State Machine Diagram from Use Case Diagrams and generate test cases. In this work, each Use Case is converted to a State Machine Diagram and then an AI planning algorithm is used to generate test cases. There are works based on interaction diagrams (Sequence Diagram, Communication Diagram) rather than Use Case Diagram. However, using an interaction diagram does not differ much from using a Use Case Diagram and its scenarios. Wu et al. [Wu 03] uses the Collaboration Diagram, Sequence Diagram and State Machine Diagram for Integration Testing. Javed et al. [Javed 07], Andrews et al. [Andrews 02], Heckel and Lohmann [Heckel 03], Pilskalns et al. [Pilskalns 03] used Sequence Diagrams and others for system-testing. Abdurazik and Offutt [Abdurazik 00] used the Collaboration Diagram for static checking and test generation. The limitation of this work is manual creation of interaction diagrams. The developers must supply the Sequence Diagram, Activity Diagram or State Machine Diagram to generate test cases.

The consistency issue is inherent in UML. Rasch and Wehrheim [Rasch 03] discuss consistency between Class Diagram and State Machine Diagram. Tsiolakis [Tsiolakis 00] discusses consistency between Class Diagram and Sequence Diagram. Gallardo et al. [Gallardo 02] discuss Sequence Diagram and State Machine Diagram, which is also discussed in this paper. However, Gallardo et al. [Gallardo 02] use a system-wide State Machine Diagram with a System behavior Sequence Diagram. The main difference from our approach is that we make a system-wide State Machine Diagram from the existing State Machine Diagram of each class and then check the consistency to it. Engels et al. [Engels 02] discuss consistency at the Meta-model level.

3 GNC (Guidance aNd Control) Missile Project

The GNC Missile Project is an on-going project in ADD (Agency for Defense Development) in South Korea. New project using Model-Driven technology is started. The first part of the project applies modeling technology to the existing source code structures, to find out where inefficiency lies and how to improve the existing structures. As of writing this paper, the first part has completed and the second part has started. The second part of the project is to generate source code directly from the improved design and to build the application with them and compare performance with that of the existing application.

The original project lacks proper documentation. Non-standard, non-UML drawings were used in the report documents and there were many confirmations that they were hard to understand. As a result, UML diagrams, which describe the current application, were made. After that, we applied our methodology of deriving system behavior and used it for the system analysis.

3.1 The Structure of GNC_SW project

GNC_SW is the program embedded in the Missile. The Missile has a simple aim. It must fly to the destination and explode where directed. We have identified the Missile Project Use Cases. There are 3 Use Cases and 10 Actors at the highest level. In the following Use Case Diagram, one of the Use Cases is shown. 8 Actors are associated with this Use Case but only 5 Actors are shown for later discussion.

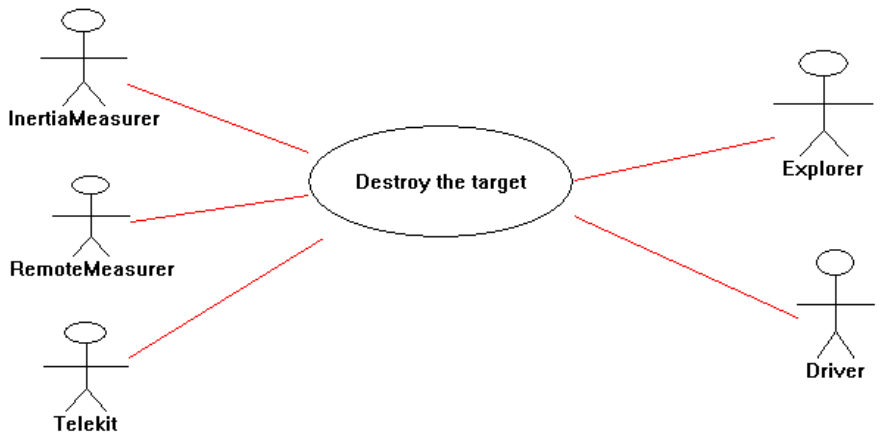

Figure 1: Use Case Diagram of GNC project

The “Destroy the target” Use Case has a primary scenario in the Sequence Diagram (Figure 2). The Ground Control sends evGccCmd – evGccEOC – evFire – evMissileAwayReady – evMissileAway in the sequence and the Missile starts flying. Whether the Missile is waiting to be fired or flying, it constantly checks the sensor

input and reports its status to the Ground Control. It is shown as a loop in the Sequence Diagram.

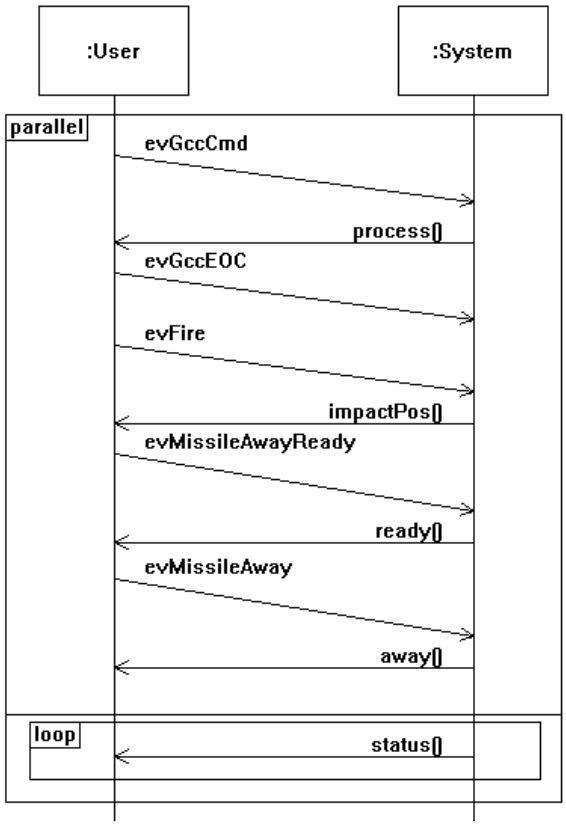

Figure 2: Use Case Scenario as a Sequence Diagram

The GNC_SW application is composed of several Tasks run in VxWorks RTOS (Real Time Operating System). The Main Task spawns GCCTask (Ground Control Communication Task), GNCTask (Guidance aNd Control Task), MIMTask (Missile-bus Interface Manager Task) and others. Most Tasks work in two ways. First, they receive a periodical event from AlarmCtrl. With this periodical event, Tasks check inputs and do other chores they need to. Some tasks report status to other tasks or to the Ground Control Center. Second, they react to specific events, such as ones in Figure 2, to perform their operations.

Tasks were identified as Active Classes in the UML Class Diagram. AlarmCtrl is the class that handles ISR (Interrupt Service Routine). There are more Tasks (thus Active Classes) in the program but we will show some of them only, because others are simplistic tasks only doing periodic jobs.

AlarmCtrl gets system clock every 100 micro-sec and sends the event to the appropriate tasks. The conceptual working of AlarmCtrl can be described in Figure 4.

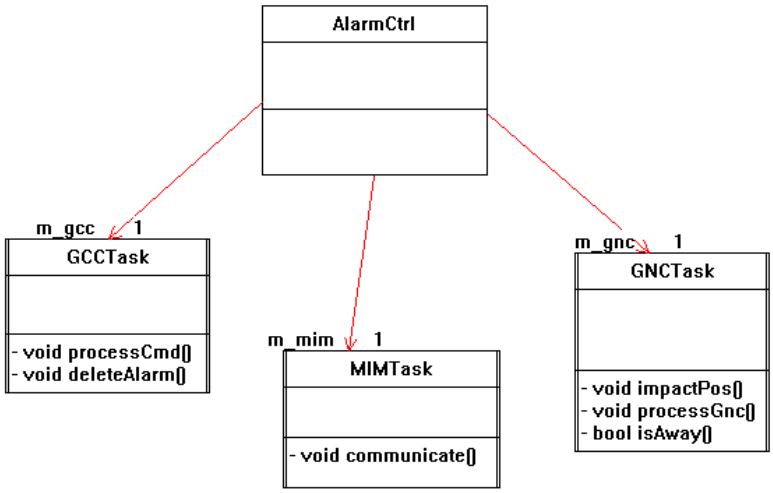

Figure 3: GNC Class Diagram

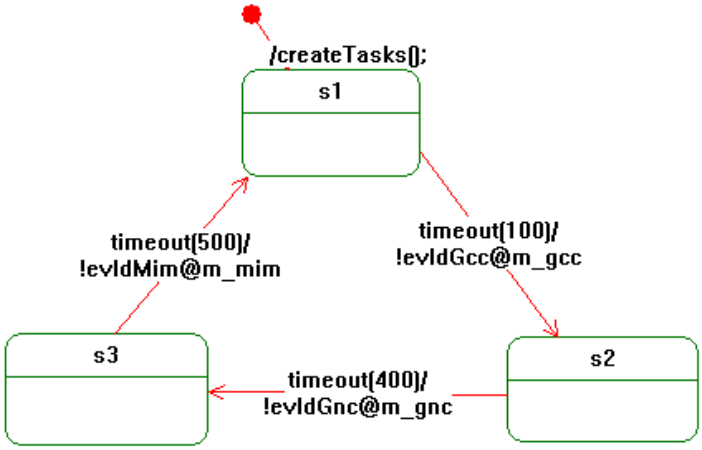

Figure 4: AlarmCtrl State Machine Diagram

GCCTask, GNCTask, MIMTask also have their own State Machine Diagram, shown in Figure 5. We have identified more classes that receive events from the aforementioned classes. We carefully categorized the classes to ones that have state-based behavior and others that are better to be portrayed in an Activity Diagram, because they are more like an algorithm. Only high level Tasks were used in the analysis we did.

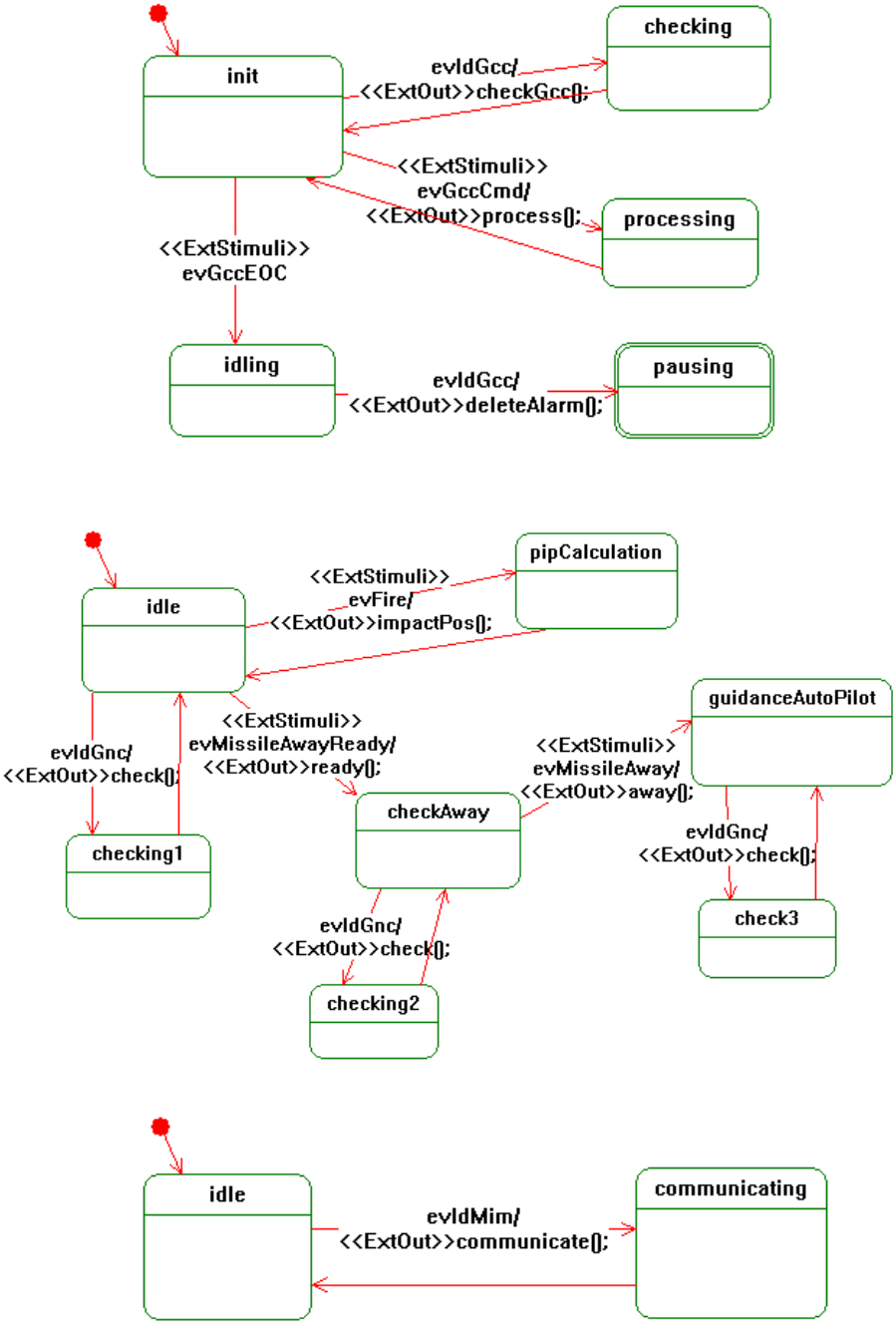

Figure 5: State Machine Diagrams of GCCTask, GNCTask, MIMTask

4 Deriving System-wide State Machine Diagram

We focus on the Behavior Diagram in this paper. State Machine Diagram and Activity Diagram are such diagrams but in UML 2.x Activity Diagram is geared to describe the flow of activities. In this paper, only the State Machine Diagram will be used to describe the behavior of the class and the system.

- State Machine Diagram and its Definition

Definition: A State Machine Diagram is a tuple SC = (S, T, s_0, t_0, F), where
S = { s_1, s_2, •••, s_m } is a finite set of states;
T = { t_1, t_2, •••, t_n } is a finite set of transitions;
s_0 is the initial state;
t_0 is the initial transition to s_0;
F = { f_1, f_2, •••, f_n } is a finite set of final states.

The Missile project GNC_SW will be used throughout to explain how the system-wide State Machine Diagram can be made automatically. Each class shall have one instance during run-time. Note that if we use a state hierarchy using a nested state or concurrent /orthogonal states, we could make the State Machine Diagrams simpler with fewer transitions. However, to proceed further, nested states and concurrent/orthogonal states need to be flattened first. ("Flattened" means that the State Machine Diagram has no nested states or concurrent/orthogonal states.) In this paper, already flattened State Machine Diagrams are used, since we are building a system-wide State Machine Diagram using the all the State Machine Diagrams in the program. Essentially, it is similar to the Model-Checking concept, so a flattened State Machine Diagram is needed to make a computation tree. Kim et al. [Kim 99] show how nested states and concurrent/orthogonal states can be removed. This results in a simple flattened State Machine Diagram.

Also note there are Stereotypes <<ExtStimuli>> and <<ExtOut>> in some of the transitions. These stereotypes will be used in the "Externalizing State Machine Diagram" step.

4.1 Merging State Machine Diagrams

The System State Machine Diagram is composed by merging State Machine Diagrams in the program. Merging is basically the cross product of states in the State Machine Diagram. Merging State Machine Diagrams can be defined in the following.

- Definition: Let SC_1 = { S_1, T_1, s_{01}, t_{01}, F_1 } and SC_2 = { S_2, T_2, s_{02}, t_{02}, F_2} be two State Machine Diagrams; the cross product of two State Machine Diagrams is defined as $SC_r = SC_1 \otimes SC_2$

SC_r = { S_r, T_r, s_0, t_0, F_r }, where
$S_r = S_1 \times S_2$ = { (s_{11}, s_{21}), (s_{11}, s_{22}), (s_{11}, s_{23}), … (s_{11}, s_{2m}), (s_{11}, s_{21}), … (s_{1n}, s_{2m}) } is a finite set of the cross product of states in two State Machine Diagrams.

$T_r = T_1 \cup T_2$

s_0 = (s_{01}, s_{02})

t_0 = transition to s_0. Initial transitions t_{01} and t_{02} are removed

$F_r = (F_{01}, F_{02})$

ⓧ has the following properties.

Rule 1) Commutative Law

sc_1 ⓧ $sc_2 = sc_2$ ⓧ sc_1

Rule 2) Associative Law

$(sc_1$ ⓧ $sc_2)$ ⓧ $sc_3 = sc_1$ ⓧ $(sc_2$ ⓧ $sc_3)$

Merging State Machine Diagrams is simple. The final product is essentially identical to the EFSM (Extended Finite State Machine) with all concurrent states removed [Kim 99]. In our case study, four State Machine Diagrams have 3, 5, 7, and 2 states each; the resultant State Machine Diagram will have 3 x 5 x 7 x 2 = 210 states and 972 transitions. Merging State Machine Diagrams can be done automatically. A simpler example using a hypothetical system is explained in Min et al [Min 09a]. Since the merged State Machine Diagram is quite complex to look at, it will not be shown here. The merged State Machine Diagram is complex, yet its inclusion of invalid transitions causes a failure to explain the system adequately. The invalid transitions are those that are never executed to go through.

4.2 Synchronization and Externalization

The synchronization concept is well explained in CSP (Communicating Sequence Process) [Hoare 87] and used in formal method tools, such as the SPIN via channel. The example in [Hartmann 00] uses synchronization as well. In one of the transitions of the State Machine Diagram of class AlarmCtrl, action '!evIdGcc@m_gcc' is shown. This means that event/signal evIdGcc will be sent to m_gcc object. In class GCCTask's State Machine Diagram, evIdGcc works as a trigger in transition. This is where synchronization comes into play. If the System Clock sends the timeout event to class AlarmCtrl's instance, it will trigger evIdGcc to class GCCTask's instance. If the merged State Machine Diagram has these transitions in sequence, synchronization takes place and two transitions can be reduced to one transition. This is explained in Figure 6. (However, this does not mean that the intermediate state can be removed safely. Due to other transitions going out or coming in to that state, the state needs to remain, unless there is no other incoming/outgoing transition.) Also note that the surviving transition has all the actions (except Sending Event section) of the removed transitions. As was stated, the synchronization can be said to sort out the valid internal transitions and eventually remove the valid transitions.

Synchronization removes some of the synchronized transitions but no state that has other transitions. It is even possible that no transition is removed, if the state has other transitions with the synchronized transition. In this case, only the first transition's destination is changed. If synchronization is applied to the case study above, the State Machine Diagram is reduced internally with the same number of states (210) and reduced number of transitions (288). The synchronized State Machine Diagram is not shown here either due to its complexity. A simpler example and its synchronized State Machine Diagram can be found in Min et al [Min 09a]. Formal definition of Synchronization is as follows.

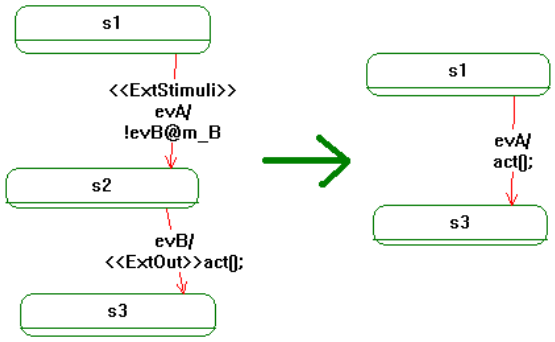

Figure 6: Synchronization

- Definition: Let Transition t = { S_{src}, S_{tar}, trigger, guard, action }
  Let f is a function synchronizing a State Machine Diagram:
  f(SC) = SC_{syn}, where SC_{syn} = { S, T′, s_0, t_0, F }
  When there are two transitions
  t_1 = { S_{src1}, S_{tar1}, $trigger_1$, $guard_1$, $action_1$ } and t_2 = { S_{src2}, S_{tar2}, $trigger_2$, $guard_2$, $action_2$ },
  if $S_{tar1} = S_{src2}$, and $action_1 \supset trigger_2$, t_1 and t_2 becomes
  t_3 = { S_{src1}, S_{tar2}, $trigger_1$, guard, $action_2$ } where guard = $guard_1 \cup guard_2$

The Synchronized State Machine Diagram is not a suitable model to describe system behavior yet. The synchronized State Machine Diagram is still too complex. This is because there are numerous invalid transitions in the State Machine Diagram. The valid internal transitions are checked and removed. However, there remain many internal – and invalid - transitions. It is the designer's job to determine whether a transition is internal or external. The Stereotype <<ExtStimuli>> is used to mark that the transition is external. The Stereotype <<ExtStimuli>> is added by the designer of the software while it is being developed. A small embedded system is probably the best to explain what <<ExtStimuli>> is. If a small system has few buttons, clicking those buttons can be mapped to each event directly. Even though there are many transitions in the system, the number of the transitions that the user directly causes is rather small compared to all the transitions in the system. The designer can add <<ExtStimuli>> Stereotype to those transitions. Transitions without <<ExtStimuli>> can be considered internal transitions which the user cannot see externally. Also the Stereotype <<ExtOut>> can be used to set the external action the tester can check. The rules to externalize the State Machine Diagram with Stereotypes are listed, as follows.

Rule 1) If there is no transition with <<ExtStimuli>>/<<ExtOut>>, the State Machine Diagram may be completely ignored.
Rule 2) If a certain state does not have the <<ExtStimuli>>/<<ExtOut>> transition (either Incoming or Outgoing), it can be removed.
Rule 3) Transition(s) without <<ExtStimuli>>/<<ExtOut>> can be removed.

- Definition: Let SC = { S, T_{nonext}, T_{ext}, s_0, t_0, F }, where T_{ext} is Transition with <<Ext>> Stereotype

Let f be a function externalizing a State Machine Diagram:

f(SC) = SC_{ext}, where SC_{ext} = { S′, T_{ext}′, s_0, t_0, F′ }

$S \supset S'$

$T_{ext} \supset T_{ext}'$

$F \supset F'$

Together with merging operator ⓧ, f has the following properties.

$f(SC_1 \otimes SC_2 \otimes SC_3) = f(f(SC_1 \otimes SC_2) \otimes SC_3)$

$f(SC_1 \otimes SC_2 \otimes SC_3 \otimes SC_4) = f(f(SC_1' \otimes SC_2') \otimes f(SC_3' \otimes SC_4'))$

The above properties are very useful for scalability. When a system is big, merging all the State Machine Diagrams leads to a very big State Machine Diagram. The usual strategy taken for scalability is "Divide and Conquer". The system is divided by several sub-systems and designed, developed, tested in sub-systems. The same approach can be applied here. The merged State Machine Diagram could be very big, depending on the size of the system. However, the system can be divided to manageable sized sub-systems. The behavior of the sub-system is independently developed and tested by the proposed method. Then, sub-system State Machine Diagrams can be merged into a system-wide State Machine Diagram and can be tested.

With those rules applied, the resultant State Machine Diagram has 210 states and 72 transitions. The number of states is the same, but now we have removed a great number of transitions. The externalized State Machine Diagram is still very complex. We have removed the internal transitions so the remaining transitions are all external. However, if we look carefully, there are states that cannot be reached from the default state. In Figure 7, the state at the bottom is one that cannot be reached from the default state. (Figure 7 is a small part of the whole diagram.). Apparently these non-reachable states may be removed safely. We can apply an optional reachability test to each state to reduce more states. This test checks whether the state can be reached from the default state. The final State Machine Diagram with reachability test is shown in Figure 8. The reachability test pseudo-algorithm is shown in Listing 1.

This final State Machine Diagram has only 27 states compared to the original 210 states and 72 transitions compared to the original 972 transitions. There is one more externalization to consider. In the State Machine Diagram of GCCTask (Figure 5), two transitions have <<ExtOut>> Stereotype in process() and deleteAlarm() action. During externalizing the State Machine Diagram, actions without <<ExtOut>> are removed and only actions with <<ExtOut>> remain, as shown in Figure 8. The Stereotype <<ExtOut>> means that it is the action the user can check from the system. That is, this is the return value we can check in the system. The user can decide pass/fail with an <<ExtOut>> action.

This State Machine Diagram in effect shows the black-box behavior of the system. All the events are from the external environment; i.e. the user, and all the actions are ones the user can check. The externalized State Machine Diagram is a reduced form

of the original State Machine Diagram. Externalization is sound, because it uses the original path without modification. Externalization is composed of two steps.

- Externalization – Transition without <<ExtStimuli>> Stereotype is removed. <<ExtOut>> Stereotype is used to determine whether to remove the action.
- Reachability Test – States that are not reachable from the default state are removed.

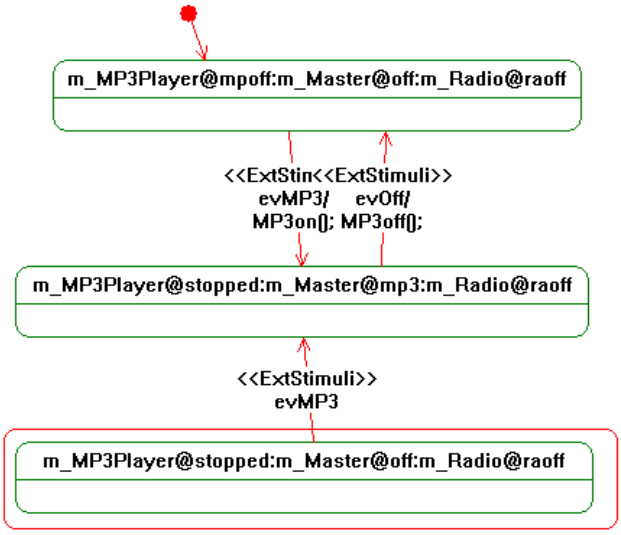

Figure 7: Samples of unreachable states from the default state

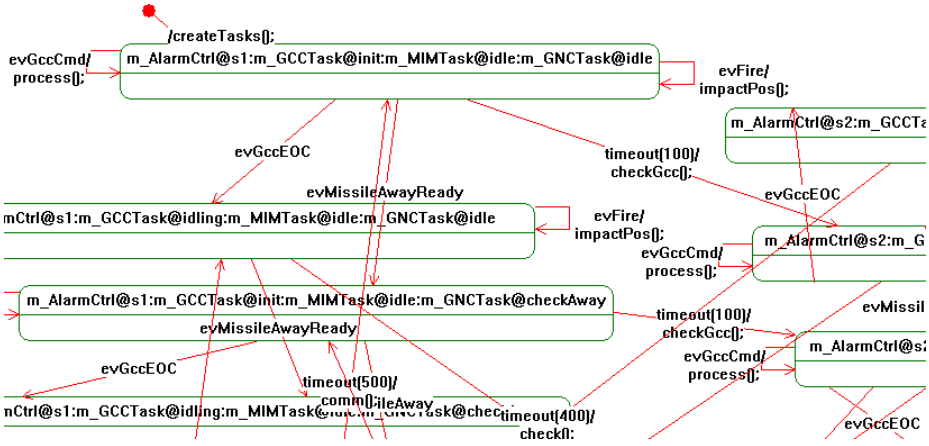

Figure 8: Externalized State Machine with RT (partial)

Listing 1: Pseudo-Algorithm of Reachability Test

```
boolean CheckReachable(State targetState,
                                    State startState) {
  // First, check if this is default state
  IF targetState is startState
    return true;
  END IF
  TransitionList listOutgoing =
                    startState.GetTransitionsFromState();
  FOR EACH Transition  tran of  listOutgoing
    m_listPath.push_back(tran);
    State nextState = tran.GetToState();
    // already went through
    IF m_listPat has nextState
      CONTINUE;
    END IF
    m_listPath.push_back(nextState);
    IF nextState is targetState
      return true;
    ELSE
      IF CheckReachable(targetState, nextState))
        return true;
      END IF
     END IF
  END FOR EACH
  return false; // no outgoing state
}
```

<<Ext>> Stereotype plays an important role. Since some transitions are external ones in the sub-system context but internal ones in system-wide context <<ExtStimuli>> is a relative term. The GNC Missile case study can be used to build a system State Machine Diagram incrementally. If the GCCTask is needed to test only, not other tasks, AlarmCtrl and GCCTask can be used to form a new sub-system State Machine Diagram. In this case, only relevant transitions have <<ExtStimuli>> Stereotype. All the transitions related with GNCTask, MIMTask, which had <<ExtStimuli>> Stereotype, now have no Stereotype. GNCTask, and MIMTask can be made as a sub-system also and sub-systems can be merged to make a system.

5 Using System State Machine Diagram

5.1 Generating Test Cases

The first thing we can do with the system-wide State Machine Diagram is to generate test cases for the system. Generating test cases from a State Machine Diagram is thoroughly studied in the academic area and already utilized in some commercial tools, such as IBM Rational Rhapsody ATG (Automatic Test Generator) and Statemate ATG. Basically, the approach consists of establishing a DAG (Directed

Acyclic Graph) from the State Machine Diagram and using a Forward/Backward search for all the paths

Kim et al [Kim 99] suggested three coverage criteria for State Machine Diagram test cases: Path coverage, State coverage, Transition coverage. Path coverage means all possible paths in the State Machine Diagram. These can easily be infinite in real systems. State coverage means that test cases cover each state at least once and Transition coverage means that test cases cover each transition at least once. Transition coverage includes State coverage. In this paper, new coverage is proposed.

Proposed coverage: ANR (All transition, No-revisit, Re-startable) coverage

- Covers all transition
- Covered transition is not visited twice (loop is covered only once)

All paths start from the default state and end in the default state or final state

Many system-wide test generation research papers use more than one diagram. In such cases, interaction diagrams (Sequence Diagram, Communication Diagram) are often used. Those diagrams are already kinds of test cases and are usually used to confine generated test cases, because automatic generation of test cases often yields too many test cases. “All path coverage” is ideal for the coverage but the number of test cases for that coverage is usually infinite. Thus “All path coverage” is impractical. Even other coverage criteria often yield too many test cases, so modified coverage from “All transition coverage” is suggested. It is vital that all transitions and all states are covered during the test. However, to reduce the number of test cases, ANR does not revisit the transition that is covered already. All test cases should start from the default state of State Machine Diagram and end in the default state or one of the final states to be able to do this. This way, another test case can start after the completed test case. In the case of the final state as an end state, the system restarts and executes the next test case.

The algorithm used here is Breadth-First search. The search starts from the default state and goes through all the paths until it finds the default (first) state again or the final state. The pseudo-algorithm is shown in Listing 2.

Listing 2: Pseudo-Algorithm of Test Generation

```
void getNextState(startState, endState, elementList) {
  TransitionList listOutgoing =
                  startState. GetTransitionsFromState();
  FOR EACH Transition tran of listOutgoing
    newElementList = elementList.copy();
    newElementList.add(tran);
    nextState = tran.getNextState();
    IF nextState is endState or nextState is final_state
      newElementList.add(nextState);
      // End of One TestCase
      m_vecTestCases.add(newElementList);
      CONTINUE; // End of this path
    END IF
      // visited before?
```

```
      boolean isBefore = newListElement.has(nextState);
      // Check is done... push it
      newListElement.add(nextState);
      IF isBefore is true
        TransitionList listOthers =
      getTransitionsFromState(allTransitions, nextState);
        IF listOthers.size is 1
          // The only path is going back.
          newListElement = null;
          CONTINUE; // end of this path
        ELSE
          boolean bFoundNext = false;
          FOR EACH Transition tran of listOthers
            State otherState = tran.nextState();
            IF newListElement has otherState
              CONTINUE; // try next iteration
            ELSE
              bFoundNext = true;
              newListElement.add(otherTran);
              newListElement.add(otherState);
              getNextState(otherState, endState,
                                        newListElement);
            END IF
          END FOR EACH
          IF bFoundNext isnot true) // Not feasible...
            newListElement = null;
            CONTINUE; // end of this path
          END IF
        END IF
      ELSE
        getNextState(nextState,endState,newListElement);
    END IF
  END FOR EACH
}
```

Table 1 shows the number of test cases generated from each State Machine Diagram. All the work is done on a Pentium 4 PC with 2.53 GHz CPU and 3 Gbyte RAM.

Table 1 shows the comparison of the original State Machine Diagram and synchronized and externalized State Machine Diagram. The original State Machine Diagram and synchronized/externalized State Machine Diagram produced more than 10,000 test cases, set as the maximum threshold of the test case generation algorithm. The actual number of test cases generated from each State Machine Diagram would be different but, since more than 10,000 test cases is impractical, no further generation is done. Clearly, more than 10,000 test cases are too many to execute, so some means of breakdown is needed. In the next section, two ways of breakdown will be used.

	nStates	nTrans	nTestCase	t(sec)	Remark
Original	210	972	> 10,000	23	Whole System
Synchronized	210	288	> 10,000	23	Whole System
Externalized with RT	27	72	> 10,000	23	Whole System
Externalized with RT	6	12	30	0.078	Whole Sys w/o timeout
Original	15	33	423	0.485	GCC Subsystem
Synchronized	15	23	78	0.219	GCC Subsystem
Externalized with RT	9	15	22	0.078	GCC Subsystem

Table 1: Comparison of Test Case Generation

5.1.1 Selective Test Case Generation (Horizontal Decomposition)

One advantage of the approach taken here is the ease of making a selection of test case generation. If the tester decides that certain transitions are not to be tested, the Stereotype <<ExtStimuli>> can be removed from the transitions. The tester can merge the State Machine Diagrams again or, one can simply disable the relevant transitions in the resultant State Machine Diagram. If timeout is not considered as an external event, the State Machine Diagram can be reduced into a State Machine Diagram with 6 states and 12 transitions. With this simpler State Machine Diagram only 30 test cases are generated. Indeed, timeout and its actions are periodic checking of the system, so they are not relevant in test cases.

Often a test case is best to be described as a UML Sequence Diagram. One of the test cases is shown in Figure 9. However, generated test cases revealed some design flaws in the existing system. The original design intention was very strict in that the event should be sent in the order, as in Figure 9. But, generated test cases show that other combinations of event order are also possible. This is not acceptable in the real operation. The design will be revised accordingly in the next version.

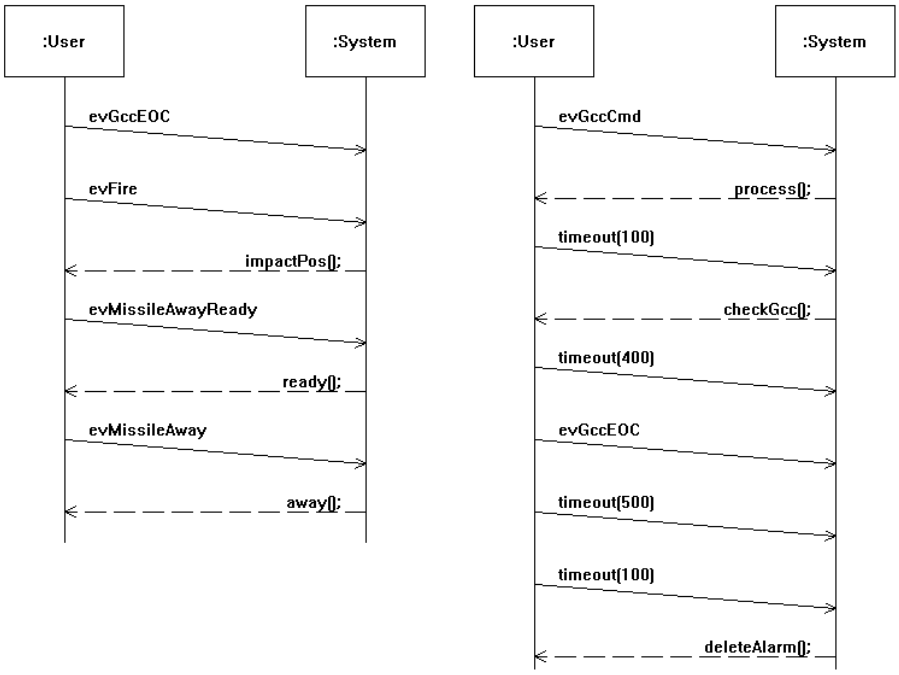

Figure 9: Generated Test Cases: Whole System & GCC Subsystem

5.1.2 Subsystem Test Case Generation (Vertical Decomposition)

The selective test case generation ability helps prevent the state explosion problem. With the approach taken here, the State Machine Diagrams must be merged first. If there are many State Machine Diagrams in the system or State Machine Diagrams are very complex, the merged State Machine Diagram will be very complex too. After synchronization and externalization, the State Machine Diagram will be much reduced but in some cases it is possible that this approach is insufficient due to the system's complexity. One way to avoid state explosion is to divide a system into small sub-systems. The approach used in this paper is useful to apply, since it is very flexible. In the sub-system, some transitions, which were originally internal transitions, will act as external transitions, since the transition which gave rise to that internal transition is in another sub-system and does not exist in the same sub-system. In such cases, it is very simple to change the transition status with the approach taken here. Adding <<ExtStimuli>> to that transition is a sufficient change to test the subsystem correctly. The GCC subsystem, which is explained and shown in the previous section, is used for the subsystem approach. The GCC subsystem produced 22 test cases from its 9 states, 15 transitions State Machine Diagram.

5.2 Deadlock detection

Deadlock is a common problem in multi-threaded program. In system-wide State Machine Diagram, it is sufficient to find the state that does not have an outgoing transition to detect deadlock. In Model-Checking, the path to lead to a particular case

is usually shown. (This is often called a counter-example.) This is simple because we have already done the reachability test and that state has passed it. To provide a counter-example, we can simply re-run the reachability test for that state alone. The high level design in GNC_SW project has four independent tasks, so deadlock checking seems necessary. However, the current design uses pre-defined timing for task running. Although Semaphore is used to make the tasks run and stop, no two tasks share a Semaphore. No deadlock is detected in system-wide State Machine Diagram, since the high level design does not have an inherent deadlock problem.

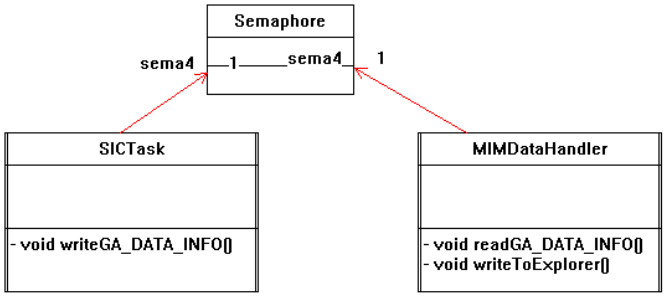

Figure 10: SICTask/MIMDataHandler/Semaphore Class Diagram

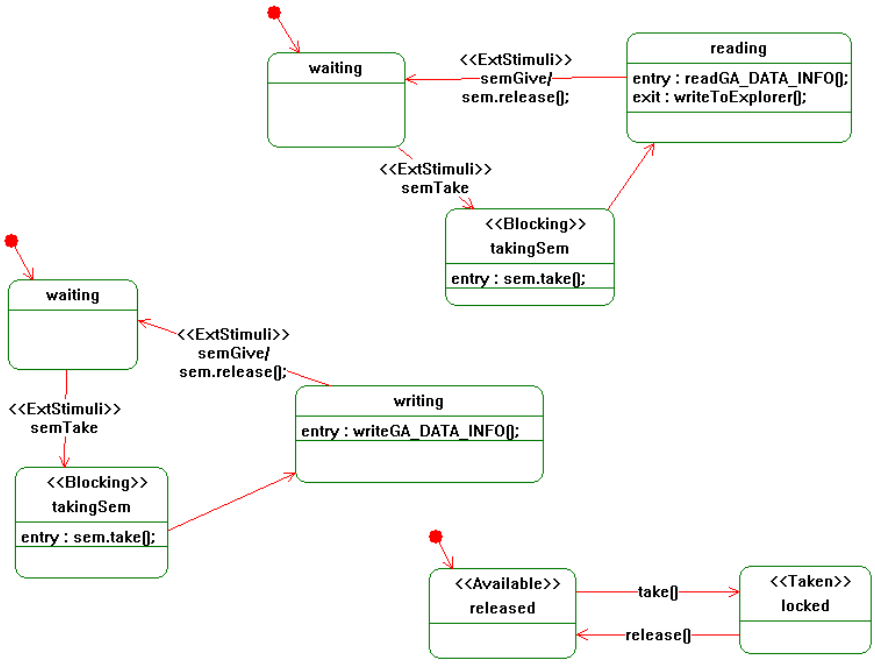

Figure 11: MIMDataHandler/SICTask/Semaphore State Machine Diagram

But, in detailed design, tasks use shared memory for communication. It is common that one task writes the data in the shared memory and later the other task reads the data from the shared memory and uses them. Semaphore is used for the protection of shared memory, and this behavior is modeled in UML diagrams again. SICTask (Subsystem Interface Control Task) writes the data in shared memory and MIMDataHandler (Missile-bus Interface Manager Data Handler) reads the data.

New Stereotypes <<Available>>, <<Taken>>, <<Blocking>> are introduced. <<Blocking>> State has a blocking function call as an entry action and NULL transition as the next transition. This <<Blocking>> Stereotype and <<Available>> Stereotype are used to determine the synchronization of transitions. Figure 12 shows an invalid transition due to a locked Semaphore. This invalid transition can be removed during the Synchronization process using Stereotypes.

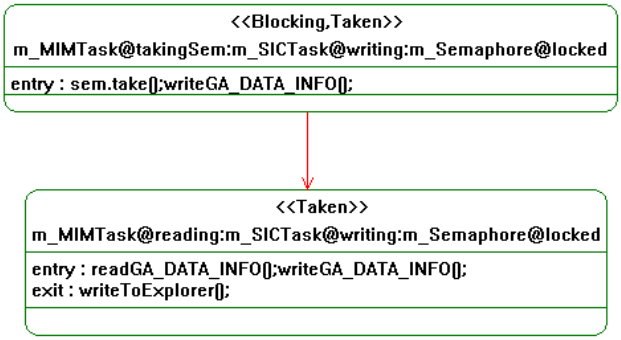

Figure 12: Invalid transition

Figure 13 shows the system-wide State Machine Diagram composed. It is a deadlock-free system.

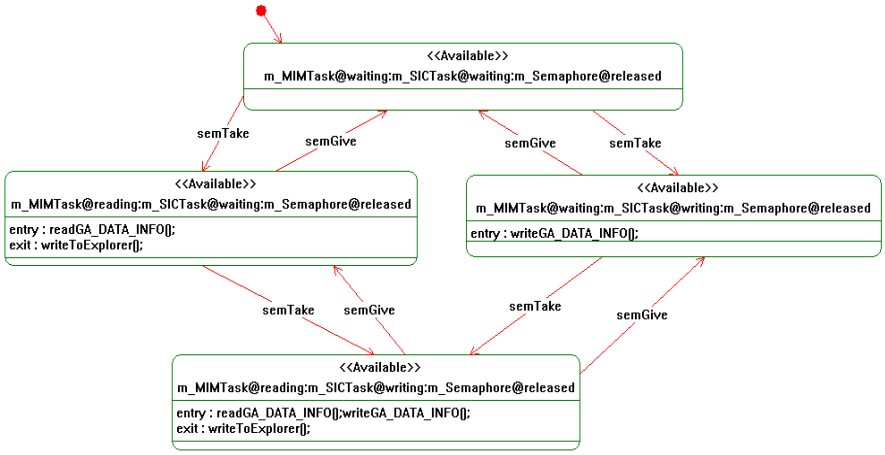

Figure 13: Deadlock free System State Machine Diagram

Since GNC_SW has simple structures only, we used the same approach to infamous "Dining Philosophers". The approach successfully identified whether the system is deadlock-free. The result is in Table 2 for the reference. The time row shows how long it takes to process the algorithm for each step.

State Machine Diagram	Two Philosophers and two Chopsticks			Three Philosophers and three Chopsticks		
	nStates	*nTransition*	*Time(sec)*	*nStates*	*nTransition*	*Time(sec)*
merged	144[a]	576	0.172	1728	10368	7.344
synchronized	144	440	0.328	1728	7944	23.563
externalized	144	216	1.547	1728	3912	11.562
ext. with RT	21	28	1.750	952	2106	383.156

Table 2: Dining Philosophers System State Machine Diagram

5.3 Consistency checking of Use Case Scenarios

A UML project often starts with Use Case analysis. A Use Case is a system's behavior. It shows what the system does and how the system interacts with the Actor. The Use Case is a black box view of the system. It does not care how the system is constructed internally. Figure 1 shows one of the Use Cases of the GNC System. It is insufficient to draw the Use Cases in Use Case analysis. Each Use Case has various scenarios. The most basic way to describe the scenarios is by using plain text, but because diagrams are easier to understand than text, diagrams can be used instead. For the description of scenarios, UML diagrams such as Sequence Diagram, Communication Diagram, and Activity Diagram can be used. Other diagrams, like State Machine Diagram, can be used as well, but the Sequence Diagram is probably the most common, since the diagram is very easy to understand. The Communication Diagram and Activity Diagram are used in the same way as the Sequence Diagram, so these two diagrams are easily interchangeable with the Sequence Diagram. In this paper, only the Sequence Diagram is considered for the scenario, but it can easily be extended to use the Communication Diagram or Activity Diagram.

The "Destroy Target" Use Case has the scenario in Figure 2. This scenario shows the basic sequence of launching the Missile. evGccCmd is sent first and evGccEOC, evMissileAwayReady, evMissileAway events are sent after that. Now that we have a correct system-wide State Machine Diagram, we can check consistency between UML diagrams. Use Case scenario, which was described as a Sequence Diagram, can be traced using the system-wide State Machine Diagram. We can check whether the scenario is correct in a step-by-step manner. A State Machine Diagram always starts with the default transition. From the default state, we can determine what kind of event is accepted and what the corresponding action is.

Figure 2 is a correct Sequence Diagram, so there will be no consistency issues. However, if we change the last action "ready()" in Figure 2 to "away()", it apparently causes inconsistency. (This is because the "ready()" action should follow after the event "evMissileAwayReady".) Hence, checking consistency reveals an error. Our implementation includes showing the error in the Sequence Diagram. In Figure 14, the result is shown. After the event "evMissileAwayReady", "ready()" action should

follow. If the scenario is incorrect with "away()", however, the result shows that "away()" is specified, but the correct one is "ready()".

If the wrong event is specified in the Use Case Scenario, our implementation shows not only that the event is inadequate, but also the list of possible candidate events that can follow in place of the incorrect event. In Figure 14, 2 events are shown after the "evMissileAway". After the "evFire" – "impactPos()", either "evMissileAwayReady" or "evFire" can be injected to the system but if "evMissileAway" is specified in the Use Case Scenario, which is incorrect, adequate events are shown. Thus, designers can understand what went wrong and how they can correct the design.

Here we assumed that Use Case Scenarios are incorrect, but what is important is the inconsistency found in UML models. It is possible that two models (Use Case Scenarios as Sequence Diagrams and State Machine Diagrams of the classes) are valid in their own contexts but could be inconsistent when they are compared together. The consistency check can be applied anytime when design is done. With the iterative approach of today's development process, when the State Machine Diagrams of classes are done, the automatic checking can follow to check if they are consistent with Use Case scenarios.

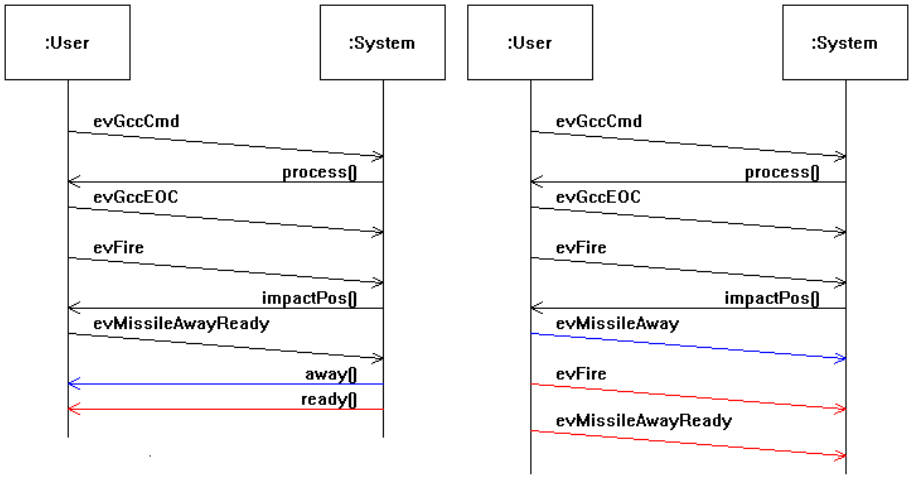

Figure 14: Inconsistent Scenarios and possible events

A simpler example using a hypothetical system with pseudo algorithm can be found in Min et al [Min 09b].

6 Conclusions

Traditionally, UML development differentiates System Analysis and Design of Software. Use Case Analysis is done at the system level and based on system analysis, more detailed design follows. Since real implementation uses detailed design, system-wide behavior is not directly related to the detailed design. This paper suggests a

method that derives the system-wide State Machine Diagram by merging the State Machine Diagrams of each class. The system-wide State Machine Diagram is made from the cross-product of all the State Machine Diagrams and then Synchronization and Externalization are applied to it. Stereotypes are used for the Externalization that is very flexible and easy to adapt.

The system-wide State Machine Diagram can be used for various purposes. First, test cases can be generated automatically using it. Testing is the most time-consuming process in the development cycle. The system-wide State Machine Diagram can generate test cases automatically and these test cases can be used in system testing. The system-wide State Machine Diagram can be used for property checking, such as deadlock detection. This is essentially the same as the Model-Checking concept. Our approach has an advantage over existing methods in that only UML is used for checking. While synchronizing, all the invalid transitions are removed and the system-wide State Machine Diagram reveals if system is deadlock-free. The system-wide State Machine Diagram can be used to compare to Use Case Scenarios for consistency checking. Since Use Case Scenarios are made prior to detailed design, designs may have inconsistencies. Use Case Scenarios are about the system behavior, so the system-wide State Machine Diagram can be used directly to check consistency.

All the research is applied to the Missile Project GNC_SW of ADD. The system-wide State Machine Diagram is made from the highest level Task designs. For the test case generation, some breakdown was inevitable due to state explosion and we have addressed how to solve it using two forms of breakdown, namely Horizontal/Vertical decomposition. Deadlock detection is performed with detailed task design and consistency check with Use Case scenarios is also performed. The legitimacy and correctness of our approach was checked easily, and the approach proved to be very useful. While the analysis is done, inefficiencies of legacy design were found, as a side effect, and these will be addressed in a future release of GNC_SW.

7 Final Remark

The GNC_SW project has a good structure for our work. Because there are three distinguished Task classes, our method was easy to apply. However, this is not the common case; there needs to be a set of general guidelines. We recommend the requirement-based sub-system approach. To apply our method, the system should not be too big. The system needs to be broken down into a certain functionality level based on requirements. If we focus on certain functionality, the number of relevant classes is relatively small and the number of State Machine Diagrams is even smaller. This is similar to the Vertical Decomposition previously mentioned in this paper. However, we recommend dividing the system by functionalities, not by the physical structures. This allows for a more logical and easier method to make a smaller system. Also, this can generate Test Cases for the functionalities and enable direct consistent checking with functional requirements. Deadlock is not a functional requirement (it is a mandatory requirement by definition) but the relevant classes and semaphores are essential for the synchronization. Thus, it is sufficient to check these classes as was demonstrated in this paper. After having a small enough system, selective externalization (Horizontal Decomposition) helps to apply our method in most cases.

Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2012R1A1A2009354). The corresponding author is Prof. Jin-Young Choi.

References

[Abdurazik 00] A. Abdurazik, J. Offutt.: "Using UML Collaboration Diagrams for Static Checking and Test Generation"; Third International Conference on the Unified Modeling Language, York, UK. 2000, Vol.1939, 383 - 395.

[Ali 07] S. Ali, L.C. Briand, M.J. Rehman, H. Asghar, M.Z. Iqbal, A. Nadeem.: "A State-based Approach to Integration Testing based on UML Models"; Information and Software Technology, 2007, Vol.49, 1087 - 1106.

[Andrews 02] A. Andrews, R. France, S. Ghosh, and G. Craig.: "Test adequacy criteria for UML design models"; Technical report, Computer Science Department, Colorado State University, February 2002.

[Briand 01] L. Briand, Y. Labiche.: "A UML-Based Approach to System Testing"; Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 2001, Vol. 2185, 194 - 208.

[Briand 02] Briand, L.C., Labiche, Y., and Soccar, G.: "Automating impact analysis and regression test selection based on UML designs"; In Proceedings of IEEE international conference on software engineering (ICSM'02) (Montréal, Canada, October 3-6, 2002). IEEE Computer Society Press, Washington, DC, 2002, 252-261.

[Briand 03] L. C. Briand, J. Cui and Y. Labiche: "Towards Automated Support for Deriving Test Data from UML State Machine Diagrams"; Proc. ACM/IEEE Int. Unified Modeling Language conference (UML 2003), Carleton University, 2003.

[Brottier 06] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon.: "Metamodel-based test generation for model transformations: an algorithm and a tool"; In Proceedings of ISSRE'06, Raleigh, NC, USA, 2006.

[Clarke 99] E. M. Clarke, Jr., O. Grumberg and D. A. Peled.: "Model Checking"; MIT Press, 1999.

[Dinh-Trong 05] T. Dinh-Trong, N. Kawane, S. Ghosh, R. France, and A. Andrews.: "A Tool-Supported Approach to Testing UML Design Models"; In Proccedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), June 2005.

[Egyed 05] Alexander Egyed.: "Instant Consistency Checking for the UML"; Proc. of the 28th International Conference on Software Engineering (ICSE), Shanghai, China, 2005.

[Elaasar 04] M. Elaasar, L. Briand.: "An Overview of UML Consistency Management"; Technical Report SCE-04-18, Carleton University, Aug. 2004.

[Engels 02] G. Engels, J. H. Hausmann, R. Heckel, S. Sauer.: "Testing the Consistency of Dynamic UML Diagrams"; Proc. 6th International Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena CA (2002).

[Fleurey 04] F. Fleurey, J. Steel, and B. Baudry.: "Validation in Model-Driven Engineering: Testing Model Transformations"; In MoDeVa'04 (Model Design and Validation Workshop associated to ISSRE'04), Rennes, France, November 2004.

[Frohlich 00] P. Frohlich and J. Link.: "Automated Test Case Generation from Dynamic Models"; Proc. 14th European Conf. Object-Oriented Programming, 2000.

[Gnesi 04] Stefania Gnesi, Diego Latella and Mieke Massink.: "Formal Test-case Generation for UML State Machine Diagrams"; Engineering Complex Computer Systems, 2004. Proceedings. Ninth IEEE International Conference

[Gallardo 02] Maria del Mar Gallardo, Pedro Merino, Ernesto Pimentel.: "Debugging UML Designs with Model Checking"; in Journal of Object Technology, vol. 1, no. 2, July-August 2002, pages 101-117.

[Gomaa 03] H. Gomaa, D. Wijesekera.: "Consistency in Multiple-View UML Models: A Case Study"; Workshop on Consistency Problems in UML-based Software Development II, 6th International Conference on the Unified Modeling Language--the Language and its applications (UML'2003), San Francisco (2003) 1-8.

[Hartmann 00] Jean Hartmann, Claudio Imoberdorf, Michael Meisinger.: "UML-Based Integration Testing"; Proceedings of the 2000 International Symposium on Software Testing and Analysis, August 2000.

[Heckel 03] R. Heckel, M. Lohmann.: "Towards Model-Driven Testing"; TACoS - International Workshop on Test and Analysis of Component Based Systems, Warsaw, April 13th, 2003,in conjunction with ETAPS 2003.

[Hartmann 04] Hartmann J., Vieira M. and Ruder A.: "UML based Test Generation and Execution"; Proc. Workshop on Software Test, Analyses and Verification, 2004.

[Hoare 87] Hoare C. A. R.: "Communicating Sequential Processes"; Prentice Hall, 1987.

[Javed 07] A. Z. Javed, P. A. Strooper, and G. N. Watson.: "Automated Generation of Test Cases Using Model-Driven Architecture"; Proceedings of the Second International Workshop on Automation of Software Test. 3. 2007.

[Kansomkeat 03]Kansomkeat, S. and Rivepiboon, W.: "Automated-generating test case using UML State Machine Diagram diagrams"; In Proceedings of the 2003 Annual Research Conference of the South African institute of Computer Scientists and information Technologists on Enablement Through Technology, vol. 47. South African Institute for Computer Scientists and Information Technologists, 296-300.

[Kandl 06] S. Kandl, R. Kirner, and P. Puschner.: "Development of a framework for automated systematic testing of safety-critical embedded systems"; In Proc. Workshop on Intelligent Solutions in Embedded Systems (WISES'06), 2006.

[Kaveh 01] N. Kaveh and W. Emmerich.: "Deadlock Detection in Distributed Object Systems"; In V. Gruhn, editor, Joint Proc. of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Vienna, Austria, pages 44-51. ACM Press, 2001.

[Kim 99] Y.G. Kim, H.S. Hong, S.M. Cho, D.H. Bae and S.D. Cha.: "Test Cases Generation from UML State Diagrams"; IEE Proceedings Software Vol. 146 No.4 1999.

[Lettrari 01] Lettrari, M., Klose, J.: "Scenario-based monitoring and testing of real-time UML models"; UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings. Number 2185 in LNCS, Springer-Verlag (2001) 317–328

[Lilius 99] J. Lilius and I. Porres Paltor.: "vUML: A tool for verifying UML models"; In Proc. IEEE International Conference on Automated Software Engineering, pages 255–258. IEEE Computer Society Press, 1999.

[Marinov 01] D. Marinov and S. Khurshid, TestEra: "A novel framework for automated testing of Java programs"; In Proc. 16th IEEE International Conference on Automated Software Engineering (ASE), San Diego, CA, Nov. 2001.

[Mikk 98] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann.: "Implementing State Machine Diagrams in PROMELA/SPIN"; In Proceedings of Workshop on Industrial-strength Formal Specification Techniques (WIFT 98), 1998.

[Milner 95] Milner R.: "Communication and Concurrency"; Prentice-Hall, 1st Edition, 1995.

[Min 09a] H.S. Min and J.Y. Choi.: "Automatic Test Generation for System using UML State Machine Diagram"; SERP '09, Las Vegas, Nevada, USA, July 2009.

[Min 09b] H.S. Min and J.Y. Choi.: "Automatic Consistency Checking of Use Case Scenarios"; SEA '09, Boston, MA, USA, Nov 2009.

[Nebut 06] C. Nebut, F. Fleurey, Y. Le Traon and J.-M. Jezequel.: "Automatic test generation: a use case driven approach"; IEEE Transactions on Software Engineering, vol. 32 (3), pp. 140-155, 2006.

[Offutt 99] Jeff Offutt, Aynur Abdurazik.: "Generating Tests from UML Specifications"; UML'99 – The Unified Modeling Language. Beyond the Standard. Proceeding of the Second International Conference, Fort Collins, CO, USA, 1999, Vol.1723, 416 - 429.

[Pilskalns 03] O. Pilskalns, A. Andrews, R. France, and S. Ghosh.: "Rigorous Testing by Merging Structural and Behavioral UML Representations"; in Proc. of the 6th International Conference on the UML, 2003, pp. 234-248.

[Rasch 03] Holger Rasch, Heike Wehrheim.: "Checking Consistency in UML Diagrams: Classes and State Machines"; in E. Najm, U. Nestmann, and P. Stevens, editors, Formal Methods for Open Object-Based Distributed Systems (FMOODS'03), volume 2884 of LNCS, pages 229-243. Springer, 2003.

[Rutherford 03] M. J. Rutherford and A. L.Wolf.: "A Case for Test-Code Generation in Model-Driven Systems"; In Proceedings of the second international conference on Generative programming and component engineering, pages 377–396, 2003.

[Tretmans 99] Jan Tretmans, Axel Belinfante.: "Automatic Testing with Formal Methods"; In EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review, Barcelona, Spain, November pp. 8-12, 1999. EuroStar Conferences, Galway, Ireland.

[Tsiolakis 00] Aliki Tsiolakis.: "Consistency Analysis of UML Class and Sequence Diagrams based on Attributed Typed Graphs and their Transformation"; ETAPS 2000 Workshop on Graph Transformation Systems (GraTra), Berlin, Germany, March 2000.

[Wang 04] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, and G. Zheng.: "Generating Test Cases from UML Activity Diagram based on Gray-Box Method"; In 11th Asia-Pacific Software Engineering Conference (APSEC 2004), pages 284–291, 2004.

[Wu 03] Ye Wu, Mei-Hwa Chen and Jeff Offutt.: "UML-based Integration Testing for Component-based Software"; The 2nd International Conference on COTS-Based Software Systems (ICCBSS). pages 251-260, Ottawa, Canada, February 2003.