Verify Behaviors of CGF with ASM

Zhang Wei, Zeng Liang
School of Computer
National University of Defense Technology
Changsha, P.R.China
{wadezhang, liangzeng}@nudt.edu.cn

Li Sikun, Xiong Yueshan
School of Computer
National University of Defense Technology
Changsha, P.R.China
{sikunli, ysxiong}@nudt.edu.cn

Abstract—**Verification is one of the most important steps in modeling and simulation. However, behavior modeling is extremely complex, especially in battlefield simulations. Till now, there are few efficient methods for behavior verification of computer generated forces. In this paper, a novel approach is proposed based on the theory of Abstract State Machines (ASM), which is usually used in the field of model-based software testing. In our work, Behaviors of CGF are described in ASML, which is a kind of specification language of ASM. With this approach, the verification of the states of CGF behaviors can be possibly achieved at the design stage, and we can improve our behavior design then.**

Keywords-Abstract state machine, AsmL, Computer generated forces, Behavior

I. INTRODUCTION

Computer generated forces [1-3] (CGF), which is one of the most important parts of the distributed virtual battlefield environment, is implemented as a computer model of soldiers or weapon platforms. It can greatly reduce the requirements of equipments and weapons. Military applications require the performance of CGF entities' behavior as realistic as possible. Therefore, it makes the behavior modeling laborious and difficult to verify. It also often lacks systematic engineering methodology, clear semantics and adequate tool support.

Abstract State Machine (ASM) theory is a formalized method for the description of software design, which support high-level semantic models and stepwise refinement approach.

We apply this theory and its method to CGF behavior modeling. And it describes and verifies the states of CGF behaviors formally at the early design stage.

II. ASM AND ASML

The concept of ASM can be traced back to the mid-1980s. The inventor of ASM, Yuri Gurevich, tried to improve Turing's thesis. He formulated the ASM Thesis: every algorithm, no matter how abstract, is step-for-step emulated by an appropriate ASM. In 2000, Gurevich axiomatized the notion of sequential algorithms, and proved the ASM thesis for them. Roughly stated, the axioms are as follows: states are structures, the state transition involves only a bounded part of the state, and everything is invariant under isomorphisms of structures. (Structures can be viewed as algebras, which explains the original name evolving algebras for ASMs.) The axiomatization and characterization of sequential algorithms have been extended to parallel and interactive algorithms.

Over the past 20 years, The method built around the notion of ASM has been proved to be a scientifically well founded and an industrially viable method for the design and analysis (verification and validation) of complex systems, which has been applied successfully to programming languages, protocols, embedded systems, architectures, requirements engineering, etc.

A. basic concepts of asm

ASM is a dynamic algebra on an alphabet Σ, while $\Sigma = \{f_1, f_2, f_3, \ldots\}$. Here, f_n is an n-ary function symbol. Function symbols are divided into static symbols and dynamic symbols. 0-ary function static symbols are called constants. It's always assumed that Σ contains Undef, False, and True. False and True are Boolean values. Undef, which is for partial functions, means a function is not defined at some point.

A sequential ASM is defined as a set of transition rules of form

if Condition then Updates

which transform first-order structures (the states of the machine), where the guard Condition, which has to be satisfied for a rule to be applicable, is a variable free first-order formula, and Updates is a finite set of function updates (containing only variable free terms) of form

$t := f(t_1, t_2, \ldots, t_n)$.

The execution of these rules is understood as updating, in the given state and in the indicated way, the value of the function f at the indicated parameters, leaving everything else unchanged. (This proviso avoids the frame problem of declarative approaches.) In every state, all the rules which are applicable are simultaneously applied (if the updates are consistent) to produce the next state. If desired or useful, declarative features can be built into an ASM by integrity constraints and by assumptions on the state, on the environment, and on the applicability of rules.

There are four types of rules: Block Rule, Conditional Rule, Choose Rule, and Simultaneous Rule.

A Block Rule R is a sequence of transition rules:

$$R_1, \ldots, R_n$$

All R_n should be executed when R is executed under the state S.

A Conditional Rule R is like this: (g is a term, R_0 and R_1 are rules)

```
if g then
    $R_0$
  else
    $R_1$
end if
```

The rule means: if the value of g under the state S is True, the updates of R under S are the same as the updates of R_0 under S; Otherwise, they are the same as the updates of R_1 under S.

A Choose Rule R has the form:

```
choose v satisfying c($v$) do
    $R_0(v)$
end choose
```

In this rule, v is a variable, c(v) is an item with v, and $R_0(v)$ is a rule. The result of this rule is uncertain. When R is executed under the state S, an element a which satisfy c(a)=True is chosen, and R_0(a) is executed; If such element doesn't exist, do nothing.

A Simultaneous Rule R is:

```
for all c($v$) do
    $R_0(v)$
end for
```

The rule means: for all elements v which satisfies c(v)=True, do $R_0(v)$; If no element satisfies the condition, do nothing.

B. *asml*

AsmL [4, 5] is an industrial-strength software specification language based on the theory of ASM. It developed by the group on Foundations of Software Engineering (FSE) at Microsoft Research, and the current version is AsmL2 (AsmL for Microsoft.NET). This version is embedded into Microsoft Word. It uses XML and Word for literate specifications. It is fully interoperable with other .NET languages. AsmL generates .NET assemblies which can either be executed from the command line, linked with other .NET assemblies, or packaged as COM components. With AsmL, A Human-readable and Machine-executable model can be created.

AsmL provides the foundations of the model-based testing tool Spec Explorer [6,7] . The Spec Explorer distribution includes the latest AsmL compiler for Microsoft .NET. Spec Explorer can also explore models written AsmL. It extends Microsoft Visual Studio for creating models of software behavior, analyzing those models with graphical visualization, checking the validity of those models, and generating test cases from the models.

Specifications written by AsmL are called executable specifications which have several remarkable features.

The executable specification describes how software components work as the traditional software specification does. But the difference between them is that the executable specification has an exclusive, certain meaning. The meaning is shown as abstract state machines, mathematical models, or run-time states.

III. MODELING CGF BEHAVIORS WITH ASML

Our CGF behavior module is built based on enhanced finite state machines (FSM). Each action has its certain execution time. Each behavior module monitors its actions, and starts, continues or terminates each state as needed.

Althrough the high-level actions are simple, CGF behaviors are very complex. Describing an entire CGF system is a huge project. Thus, we chose a representative action "GoToTarget" to show how it is modeled by AsmL.

In this action, an agent is expected to find a path to reach the target, and avoid all the obstacles on the way. If the target is unreachable finally, then it should give up. Figure 1 shows how the action works.

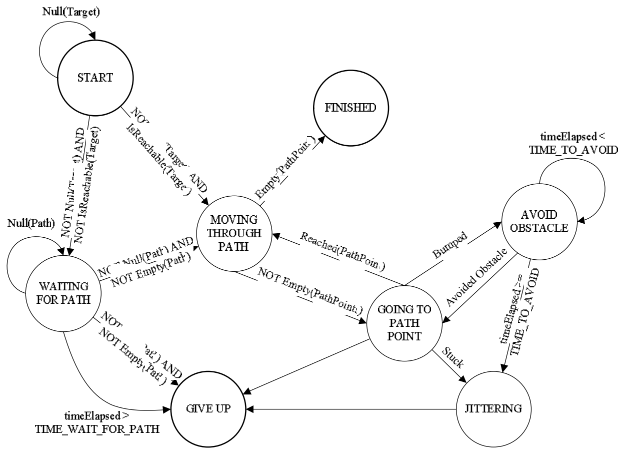
Null(Target)
START
FINISHED
NOT IsReachable(Target) AND
NOT Null(Target) AND NOT IsReachable(Target)
Empty(PathPoint)
timeElapsed < TIME_TO_AVOID
Null(Path)
MOVING THROUGH PATH
AVOID OBSTACLE
NOT Null(Path) AND NOT Empty(Path)
Reached(PathPoint)
Bumped
WAITING FOR PATH
NOT Empty(PathPoint)
Avoided Obstacle
GOING TO PATH POINT
timeElapsed >= TIME_TO_AVOID
NOT Null(Path) AND NOT Empty(Path)
Stuck
timeElapsed > TIME_WAIT_FOR_PATH
GIVE UP
JITTERING

Figure 1. Action "GoToTarget"

We describe this action of an agent with Spec Explorer. Part of AsmL codes are presented in straw yellow regions. We defined eight states as follows:

```
namespace CGF

enum AgentState
  START
  WAITING_FOR_PATH
  MOVING_THROUGH_PATH
  GOING_TO_PATH_POINT
  AVOID_ABSTACLE
  JITTERING
  FINISHED
  GIVE_UP
```

Null(X), IsReachable(X), Reached(X) and Empty(X) are defined as predicates, while Stuck and Bumped are Boolean variables.

Target, TimeElapsed, PathPoints are variables. TIME_TO_AVOID, TIME_WAIT_FOR_PATH and TIME_RESTART are constant values.

State START, FINISHED, and GIVE UP are end states, FINISHED means the target is successfully reached, while GIVE UP means failed.

```
enum NavPoint
  PNull // Null path
  P0    // start point
  Pt    // target point
  PObs  // obstacle

var NullTarget as Boolean = false
var IsReachableTarget as Boolean = false
var Path as Set of NavPoint = {}
var timeElapsed_wait as Boolean = false
var timeElapsed_restart as Boolean = false
var timeElapsed_avoid as Boolean = false
var myState = START
```

The agent begins with the START state, and the target is set by the sensor. If no goal is set (Null (Target)==True), then the agent will do nothing.

If the target can not access directly, the module will request a path for it, and turn into the WAITTING FOR PATH state. The agent starts MOVING THROUGH PATH when a path is received.

At the state WAIT FOR PATH, the module sends a GetPath command, and wait for the path information. If the time Elapsed exceeds TIME_WAIT_FOR_PATH, The agent will give up(state GIVE UP).

```
[Action]
WaitingForPath()
  require myState = START or myState =
WAITING_FOR_PATH
  myState := WAITING_FOR_PATH

[Action]
GiveUp()
  require myState = WAITING_FOR_PATH or myState =
GOING_TO_PATH_POINT or myState = JITTERING
  match myState
    WAITING_FOR_PATH:
      myState := GIVE_UP
    GOING_TO_PATH_POINT:
      myState := GIVE_UP
    JITTERING:
      myState := GIVE_UP
```

We use a set of navigation points to present a path. The return of GetPath command is a list of navigation points. If the path does not exist, then the it is Null.

```
[Action]
GoToPathPoint()
  require myState = WAITING_FOR_PATH or myState =
AVOID_OBSTACLE
  myState := GOING_TO_PATH_POINT
```

If the agent has visited all the navigation points of a path, then it reached the state of FINISHED, and the action is successful.

```
[Action]
Finished()
  require myState = MOVING_THROUGH_PATH
  myState := FINISHED
```

If there are still navigation points, it enters the state GOING TO PATH POINT. If it meets obstacles, it enters the state AVOID OBSTACLE. If the agent has failed to avoid obstacles, it enters the state JITTERING. In this state, the agent will try some random actions to leave the place where it stays. Then, the agent enters GIVE UP state. This is because it has moved for a while and should recalculate the path from the START state.

```
[Action]
AvoidObstacle()
  require myState = GOING_TO_PATH_POINT or myState
= AVOID_OBSTACLE
  myState := AVOID_OBSTACLE

[Action]
Jittering()
  require myState = GOING_TO_PATH_POINT or myState
= AVOID_OBSTACLE
  myState := JITTERING
```

Three properties are defined for us to retrieve the state of the state machine.

```
property CGFTarget as Boolean
  get
    return not NullTarget

property IsReachable as Boolean
  get
    return IsReachableTarget

property CGFState as AgentState
  get
    return myState
```

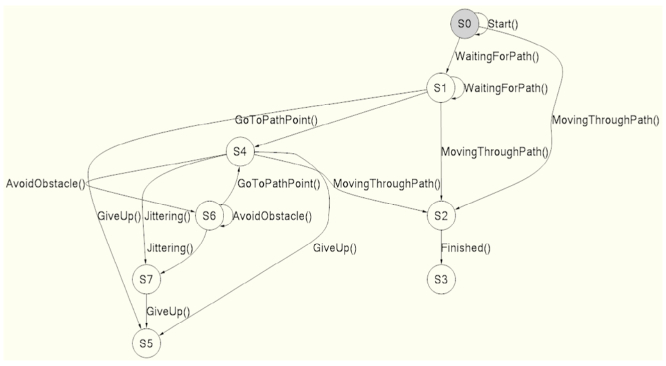

Figure 2. FSM diagram generated by Spec Explorer

After describing the action with AsmL, set "Exploration Goal" to "FSM Generation", and run this code. We will the result as shown in Figure 2.

By comparing Figure 1 and Figure 2, we can see that two finite state machine diagrams are nearly the same. It is easily to verify the design of CGF behavior module with AsmL.

IV. CONCLUSION

Models built by AsmL language can run as simulations of the systems they describe. This feature can help us check the completeness of our design before implementation. It can also help us check our design and implementation during the implementation stage.

Finite state machine provides a useful solution for the CGF behavior module, while modeling and analyzing with AsmL language helps find design flaws, and verify the completeness of the model. It also can be used to improve the design of CGF behavior module.

REFERENCES

[1] GUO Qi-sheng; YANG Li-gong; YANG Rui-ping; XU Ru-yan; DONG Zhi-ming. An Introduction to Computer Generated Forces[M], Beijing: National Defence Industry Press, 2006. (in Chinese)

[2] ZENG Liang; ZHENG Yi; LI Si-kun. Behavior Model of Computer Generated Forces Based on Cybernetics, Journal of System Simulation [J], 2005.17(4), PP773-775. (in Chinese)

[3] ZHENG Yi. The Research of Behavior Modeling and Implementation Technology of Computer Generated Forces[D], Changsha: National University of Defense Technology, 2003. (in Chinese)

[4] Foundations of Software Engineering, Microsoft Research.Abstract state machine Language[EB/OL]. http://research.microsoft.com/fse/AsmL.

[5] Gurevich Y. Evolving algebra 1993: Lipari guide[C]. Oxford University Press,1995, PP9-36.

[6] Veanes M; Campbell C;Grieskamp W; Schulte W; Tillmann N; Nachmanson L. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer[C]. 4949 of LNCS, 2005, PP39-76.

[7] Barnett M, Schulte W. Runtime verification of . NET contracts[J]. Journal of Systems and Software,2003,65(3):199~208.