

Effective Mapping for Modeling State Machines in Round-trip Engineering

Jin Xin, Jin Yuanping

Abstract: We study the problem of creating a round-trip 1 to 1 mapping between Unified Modeling Language (UML) state chart diagram and its implementation. This mapping can be used to construct a state machine by generating and managing a series of executable code from the UML state chart diagram and at the same time to support rebuilding state chart from the generated codes, thus to provide the UML tool with the capability of synchronizing its model according to modification in code part in real-time.

Key words: mapping, hierarchical state machine, UML, round-trip engineering

1 Introduction

Many concurrent, distributed and realtime applications has to tightly co-work with other objects or hardware, which are called service providers. A service is formally specified by a set of primitives (operations) available to service users (applications). These primitives describe some action or report on an action taken by a peer component/entity to be performed. The service primitives can be classified into four categories: request, indication, response and confirm[1]. It is natural to use state machines to modeling these applications, as an application that must sequence a series of actions, or handle inputs (responses and indications) differently depending on what state it's in, is often best implemented as a state machine.

State machine modeling technology for these software developments provides users a visual, dynamic view in design and implementation steps, greatly improving development efficiency and curtailing development cycles. Through this modeling, we can mainly benefit in two aspects: readability, which enables a clear system description by capturing the internal structure and external behaviors of a system; reusability, which makes a system's code be easily reused in some other similar ones.

While there are already many good UML tools such as Rational Rose, VisualState and others, this paper proposes a new feature of round-trip engineering support. An important rule in software design is that no design remains unchanged. This is as true for small systems as it is for large systems. During development, the design structure defined in the UML model does undergo changes to incorporate physical differences in implementation that may not have been envisaged during design. It becomes very difficult to keep the design of the system updated with the changes in the source code.

Most of the preceding related work has been focused on UML diagram drawing and code generation, or, in other word, one-way mapping. We consider two-way approach to mapping between state chart and executable source code. Our solution tends to enforce the UML tool's capability of synchronizing its model according to modification in code part in real-time. For example, developers can visually do a drag-and-drop operation in state chart diagram and subsequently the source code would change as well; to the contrary, developers may directly rectify the code part while modification will be automatically shown in state chart diagram. It can be seen that the solution serves as a bridge between state chart diagram and source code.

Our study is mainly based on our open source project, UML StateWizard [3], which brings out an alternative view in rapid development based on round-trip engineering theory.

- StateWizard aims at not only being a model or code generation tool, but also being a full-featured UML dynamic model tool with reverse engineering and round-trip engineering features. It models program with the State Charts or the State Tree, on the other hand, it synchronizes the model with the changes in the application code in the stages of software design and coding;
- Just like Visual C++ ClassWizard, StateWizard runs inside the popular IDE (integrated development environment). No need to switch between tools for design/development, while many other UML model tools run stand alone;
- Rapidly builds applications with state machine based frameworks and efficiently codes state charts directly in platform-independent standard C/C++;
- Provides state tracking, simulation and debugging for embedded system development.

In this paper we study the state modeling problems related with mapping, in particular, the problem of efficient generating code , given a mapping between the state machines and the source code, and of rebuilding state machines through this mapping.

While the techniques we describe are, at a high-level, intuitive, the main contributions of the paper are the techniques we developed to apply mapping between state chart and source code to rapid development, and the experimental study of their practical impact. All put together, our mapping techniques speed up application development, thereby enabling efficient development on projects of significant size.

The paper is organized as follows: Section 2 presents in more detail concepts of Hierarchical state machine and an overview of traditional approaches. Section 3 provides an explanation of our mapping solution used in StateWizard project. Section 4 presents an experiment through our solution and Section 5 concludes.

2 Preliminaries

2.1 Concepts of Hierarchical State Machine

In general, a hierarchical state machine (HSM)[4] is any device that stores the status of something at a given time and can operate on input to change the status and/or cause an action or output to take place for any given change. A hierarchical state machine should include the following elements:

- A set of hierarchical state
- An initial state or record of something stored somewhere
- A set of input events
- A set of output events
- A set of actions or output events that map states and input to output (called a state event handler)
- A set of actions or output events that map states and inputs to states (called a state transition)

2.2 Overview of Traditional Approaches

There has been considerable work on implementing a hierarchical state machine in programming languages. The most common technique to implement a state machine is the doubly nested switch statements with a “scalar variable” used as the discriminator in the first level of the switch and event-type in the second level [4][5]. This works well for classical "flat" state machines and is widely employed by automatic code synthesizing tools. Manual coding of entry/exit actions and nested states is, however, cumbersome, mainly because code pertaining to one state becomes distributed and repeated in many places, making it difficult to modify and maintain when the topology of state machine changes.

Another technique is object-oriented "State" design pattern, based on delegation and polymorphism [4][5]. States are represented as subclasses implementing a common interface (each method in this interface corresponds to an event). A context class delegates all events for processing to the current state object. State transitions are accomplished by changing the current state object (typically re-assigning one pointer). This pattern is elegant but is not hierarchical. Accessing context attributes from state methods is indirect (cannot use an implicit pointer) and breaks encapsulation. The addition of new states requires sub-classing and the addition of new events requires adding new methods to the common interface. Also, it might suffer from efficiency problems.

3. Mapping Solution in StateWizard

Our implementation of the HSM pattern is, to some degree, through action-state tables containing typically sparse arrays of actions and transitions for each state. Actions (including entry/exit, state reactions, and actions associated with transitions) are most commonly represented as pointers to functions. Representing state hierarchy in a flat action-state table is sometimes cumbersome. Considering easy-to-use problem, we purposely define a series of macros as state machine mapping data. This makes it possible for users to touch the state machine's building task without any pre-knowledge on the internal implementation. On the other hand, these mapping data will be parsed by StateWizard to construct state machines when source code files are loaded, modified.

Our mapping data are partitioned into: state enumeration declaration, event handler declaration, state event handler table, state tree definition and application variable definition. The first two components are declarations used in header files while the remained ones are in source files. As the way ClassWizard in Visual C++™ does, these mapping data need not manually created but done automatically by StateWizard. Furthermore, manual modification in code part is permitted if it conforms to the syntax of mapping data.

Figure1 and Figure2 show the *Player* application state chart/tree. The top super state of a state machine state stands for an application, which encapsulates a group of children states but has no parent. *PowerDown* and *PowerUp* states are brothers while *Playing* and *Pause* states are children of *Powerup*. The state chart and the state tree can help you create and manage the state machines for your program. You can construct state hierarchy, define event handlers and navigate through your source files, and more.



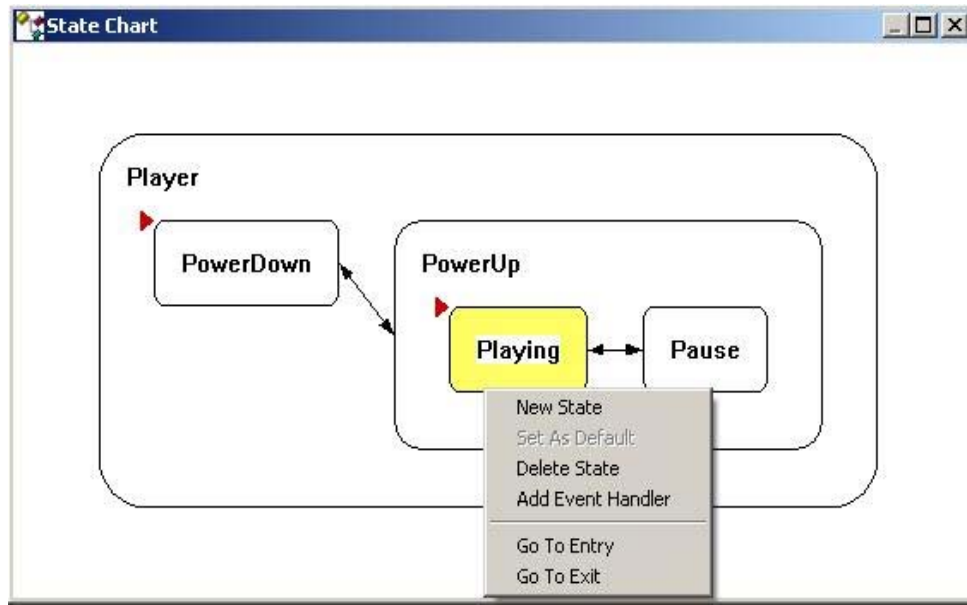


Figure 1: The *Player* application state chart

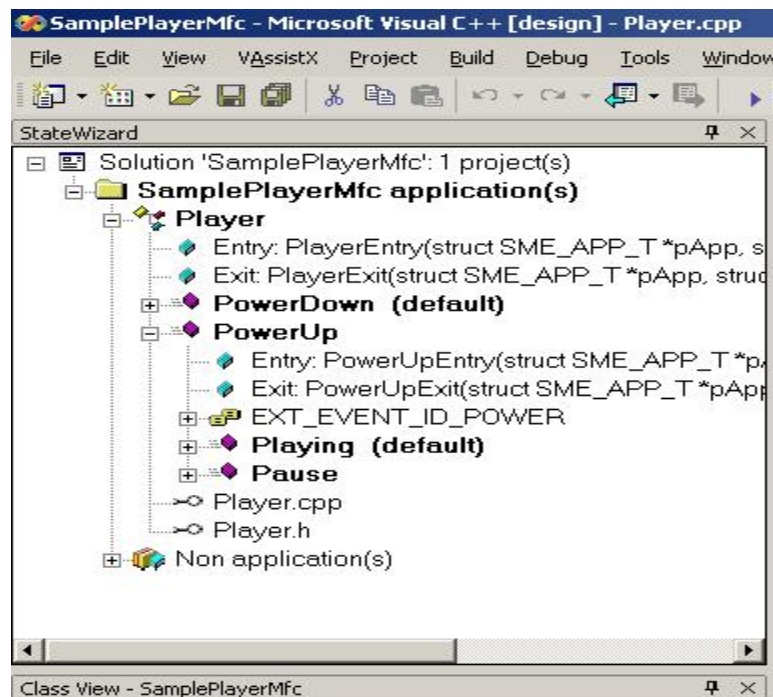


Figure 2: The *Player* application state tree

3.1 State Enumeration and Event Handler Functions

These two structures are used in header files to declare states and their handler function respectively before we define them. As we do to keep a good style in popular programming languages like C/C++ and Java, we should declare variable before we use it. State enumeration

declaration intends to declare each state in the state machine. For example, we declare the *Player* application states and event handlers as below:

```
SME_BEGIN_STATE_DECLARE(Player)
    /*{{SME_STATE_DECLARE(Player)*/
    SME_STATE_DECLARE(Player)
    SME_STATE_DECLARE(PowerDown)
    SME_STATE_DECLARE(PowerUp)
    SME_STATE_DECLARE(Playing)
    SME_STATE_DECLARE(Pause)
    SME_MAX_STATE(Player)
    /*}}SME_STATE_DECLARE*/
SME_END_STATE_DECLARE

/*{{SME_BEGIN_EVENT_HANDLER(Player)*/
int PlayerEntry(struct SME_APP_T *pApp, struct SME_EVENT_T *pEvent);
int PlayerExit(struct SME_APP_T *pApp, struct SME_EVENT_T *pEvent);
int OnPowerDownEXT_EVENT_ID_POWER
    (struct SME_APP_T *pApp, struct SME_EVENT_T *pEvent);
...
/*}}SME_END_EVENT_HANDLER*/
```

3.2 State's Event Handler Table

This structure offers a whole definition of event handlers for each state. As section 3.1 illustrated, when an event is dispatched, state machine would respond by executing corresponding event handler. In addition, such event handler can be partitioned into two parts: entry/exit handler; normal event handler. An entry/exit handler has to be performed at the time the state is entered/exited. However, a normal event handler would be activated only when its specific event is dispatched. In our mapping, the former one is of the form: SME_STATE_ENTRY/EXIT (handler name); the latter one is of the form: SME_ON_EVENT (event name, handler name, belonged state name). For example, we define the even handler table of the *PowerUp* state as below:

```
SME_BEGIN_STATE_DEF(Player,PowerUp)
    /*{{SME_STATE_DEF(Player,PowerUp)*/
    SME_STATE_ENTRY_FUNC(PowerUpEntry)
    SME_STATE_EXIT_FUNC(PowerUpExit)
    SME_ON_EVENT(EXT_EVENT_ID_POWER,OnPowerUpEXT_EVENT_ID_POWER,PowerD
own)
    /*}}SME_STATE_DEF*/
SME_END_STATE_DEF
```

3.3 State Tree

This structure mainly provides the profile of state tree/chart. It records critical information (including state name, event handler and relationship among them) to draw state chart or rebuild state tree. Modification in this code part will directly reflect in state chart/tree while changes in state chart/tree would cause automatic rectification in this structure as well. Elements in this structure are states under the dedicated application node, which are of the form SME_STATE(application name, state name, parent state name, default child state name). For example, we define the *Player* state tree as below:

```
SME_BEGIN_STATE_TREE_DEF(Player)
```

```

/*{{SME_STATE_TREE_DEF(Player)*/
SME_STATE(Player,Player,SME_INVALID_STATE,PowerDown)
SME_STATE(Player,PowerDown,0,-1)
SME_STATE(Player,PowerUp,0,Playing)
SME_STATE(Player,Playing,PowerUp,-1)
SME_STATE(Player,Pause,PowerUp,-1)
/*}}SME_STATE_TREE_DEF*/
SME_END_STATE_TREE_DEF

```

3.4 Application Variable Definition

We regard a state machine as an application. In our mapping, we declare an application with a structure in type of SME_APP_T, which is of the form SME_APPLICATION_DEF (application, “application name”). For example, we define the *Player* application as below:

```
SME_APPLICATION_DEF(Player, "Player")
```

4. Experiments

Based on the mapping technology illustrated above, we have worked out a solution to produce 1-to 1 mapping between source code and state chart/tree in state machine. State tree could be built by analyzing source code, and meanwhile, we could also construct state machines through state tree. Such round trip way also happens on state chart. It is easy to draw state chart according to state tree and we could also build state machines by state chart.

Using StateWizard, we developed a mobile phone embedded system program[3]. After embedded systems' simulating and debugging in Visual C++ developer studio, we could move program to a destination working environment with little or no extra investment of effort.

5. Conclusion

This paper presents an effective round-trip 1 to 1 mapping solution between state chart and its implementation of state machine. Our techniques can be applied to various application development based on state machine including platform-independent embedded systems, Win32 or WinCE. Therefore, it will greatly improve the productivity in software development and consequently decrease the cycles in development. Moreover, we have developed StateWizard using our technology to implement round-trip engineering, enabling the UML tool to synchronize the model with the changes in the application code.

Reference

- [1] Andrew S.Tanenbaum. Computer Networks
- [2] Hei Wei, Jin Yuanping, Jin Xin. Design and Application of Reusable Software Framework Based on Hierarchy State Machine Computer Application and Software, to be published in 2006
- [3] Jin Xin. UML StateWizard[Z] .<http://www.intelliwizard.com/> 2006-2
- [4] Samek M. Practical State charts in C/C++: Quantum Programming for Embedded Systems [M]. CMP BOOKS, 2002.
- [5] Barr, Michael. Programming Embedded Systems in C and C++. Oreilly & Associates[M], 1999
- [6] Duby, Carolyn. Class203: Implementing UML state chart diagrams. Proceedings of Embedded Systems Conference[C], Fall, San Francisco, 2001
- [7] Egon Borger, Alessandra Cavarra, Elvinia Riccobene. Modeling the Dynamics of UML State Machines