Information and Software Technology 55 (2013) 1726–1740

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

An object-oriented implementation of concurrent and hierarchical state machines

Volker Spinke

Parkstraße 8, 65439 Flörsheim am Main, Germany

ARTICLE INFO

Article history:
Received 1 October 2012
Received in revised form 12 March 2013
Accepted 16 March 2013
Available online 27 March 2013

Keywords:
State machines
UML statecharts
State pattern
Double-dispatch
Code generation
Design pattern

ABSTRACT

Context: State machine diagrams are a powerful means to describe the behavior of reactive systems. Unfortunately, the implementation of state machines is difficult, because state machine concepts, like states, events and transitions, are not directly supported in commonly used programming languages. Most of the implementation approaches known so far have one or more serious drawbacks: they are difficult to understand and maintain, lack in performance, depend on the properties of a specific programming language or do not implement the more advanced state machine features like hierarchy, concurrency or history.

Objective: This paper proposes and examines an approach to implement state machines, where both states and events are objects. Because the reaction of the state machine depends on two objects (state and event), a method known as double-dispatch is used to invoke the transition between the states. The aim of this work is to explore this approach in detail.

Method: To prove the usefulness of the proposed approach, an example was implemented with the proposed approach as well as with other commonly known approaches. The implementation strategies are then compared with each other with respect to run-time, code size, maintainability and portability.

Results: The presented approach executes fast but needs slightly more memory than other approaches. It supports hierarchy, concurrency and history, is human authorable, easy to understand and easy to modify. Because of its pure object-oriented nature depending only on inheritance and late binding, it is extensible and can be implemented with a wide variety of programming languages.

Conclusion: The results show that the presented approach is a useful way to implement state machines, even on small micro-controllers.

1. Introduction

Finite state machines are a clear and concise way to describe the behavior of a system reacting to external events. They are a well-known and widely used technique to describe the dynamics of control systems, protocols or graphical user interfaces. A state machine comprises all permitted states of a system and the allowed transitions between them. Transitions are triggered by events and can be guarded by a condition.

To make the description even more expressive, hierarchies of states have been introduced, which leads to hierarchical state machines. In some cases, actions are to be performed in parallel, which leads to concurrent hierarchical state machines. The paper of Harel [1] gives an introduction to the concepts.

The UML [2] has become the most widespread modeling language used today. It provides a state machine diagram which is a graphical representation of a state machine. The UML state machine diagram combines Mealy and Moore machines. Actions depend on both the active state of the system as well as the triggering event and are associated with the transition from one state to the subsequent state, as in Mealy machines. Additionally, it is possible to define entry and exit actions, as in Moore machines. UML state machine diagrams also allow the hierarchical nesting of states. With this features, UML is capable of modeling a large range of state machines, from simple to very complex.

State machines are an important, not to say essential, way to describe the behavior of reactive systems. They are widely used to implement the control logic of all kinds of software – on a small micro-controller as well as in a large server application.

Unlike classes and objects, current mainstream programming languages, like C++, Delphi or C#, do not support state machines directly. What we are looking for, is a way to implement state machines, which is universally applicable, independent of a special programming language, shows sufficient performance and enables us to make use of the more advanced features like nesting, concurrency and history as well as the advantages of object-orientation.

E-mail address: vs@spinke.de

0950-5849/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2013.03.005

In addition to this, we want an approach to support the UML semantics.

The search for this proposal forms the outline of the paper: at first we look at the traditional approaches and find out, that they have some limitations that make them suboptimal for the intended purpose. The next step is to clarify the requirements of a suitable approach in more detail. As a result of these considerations, we conclude, that applying the well-known object-oriented pattern named double-dispatch to implement state machines should be the solution we are looking for. Then we have a look at the literature to verify this.

Unfortunately, there were no papers found that describe how to implement state machines with the double-dispatch approach. Furthermore, there were no papers found that suggest other approaches that fulfill the requirements set before. Because of this unexpected result, this paper was written to fill this gap.

In the later sections, we describe how to apply the double-dispatch approach to the implementation of state machines and compare it with the traditional approaches. In order to have one representative for each of the traditional approaches, visualSTATE [3] (nested switch/if statements approach, state-event-table approach) and the modified approach of Niaz and Tanaka [4] (based on the state pattern by Gamma et al. [5]) are included in the comparison. In addition to these, the Boost Statechart Library [6] was chosen as a representative of language specific approaches. The Boost Statechart Library is based on lists.

The paper ends with a conclusion and an outlook on the future work.

2. Traditional approaches

In the past, a confusing large number of implementation proposals have evolved. This makes it difficult for a user to choose an appropriate one. At a closer look, there are many similarities, because they basically originate from one of three traditional approaches.

2.1. Nested switch/if statements

The most simple and straightforward approach is to nest two switch statements using scalar variables to represent the states and events. The outer switch statement e.g. selects between different states and the inner one selects between the possible events in this state. The same result can be achieved with if-statements too. Often both are combined: the outer selection is a switch-statement and the inner selection is done by if-statements.

This works well for small and simple state machines but gets cumbersome and confusing very fast, as the number of states and events grow. Besides this, the run-time heavily depends on the way how the compiler translates the switch statement into machine code. Switch statements can be implemented as a series of if statements or by using a jump table. If a series of if statements is used, the runtime is not constant but depends on the active state and the event to be processed and degrades with the number of cases. If a jump table is used, the implementation is similar to the state-event-table approach, described in the next section.

2.2. State-event-table

A more sophisticated approach is to store the transition information in a table. One dimension of the table represents all possible states, the other one all possible events. Using contiguous numbers for both states and events as an index to the table, it is easy to look up the action to perform.

With pointers to functions as table data, the execution time of this approach is fast and does not depend on the size of the table. The run-time for loading the pointer from the table is constant. Nevertheless, this advantage is spoiled to some extend, because in most applications, external events must be mapped to contiguous numbers to access the table. This introduces a search algorithm again, which increases the complexity.

The table approach too gets cumbersome and confusing as the number of states and events grow. The matrix gets large, but is usually sparse. Initialization of the table is complicated and prone to errors if done manually. Nesting is possible, but tedious to implement.

Automatic code generation can overcome these limitations, but usually makes the resulting code practically impossible to read for a human programmer. This aspect is especially important during debugging.

2.3. State pattern

The state pattern published by Gamma et al. [5] is a well-known object-oriented approach for coding state machines. The basic idea is to implement each state as a separate class and each event as a method of this state class. The invocation of a concrete method is done by delegation and late binding.

The state machine is represented by an object which offers methods for all supported events. The methods themselves delegate a call to a local state object. Now, it is easy to change the behavior of the state machine reacting to an event by simply exchanging the state object. Each state object is free to implement the event methods in a different way.

The state pattern has some nice advantages: the execution time is constant (action execution is not taken into account), due to the late binding. State-specific behavior is localized in a single class. This eases debugging and maintenance.

But there is also the other side of the medal: events must be mapped into a method call, which often requires a switch statement or search algorithm again. Further more, the state pattern requires some discipline from the developer and needs a lot of code to write. Changes to the state machine can affect quite a few classes. As van Gurp and Bosch [7] outline, the disadvantages of the state pattern mainly result from the fact, that the only concept explicitly represented is the state. All other elements of a state machine (events, transitions, etc.) are modeled merely implicitly.

Unfortunately, the original state pattern as described in [5] lacks some of the more advanced features of UML state machine diagrams: it is not hierarchical nor does is tell us how to implement entry and exit actions, concurrent states or states with history. Yacoub and Ammar [8,9] present a set of patterns, that extend the state pattern with these features. Also Niaz and Tanaka [4] present an extension to the state pattern. Adamczyk [10] provides an anthology of 23 state machine design patterns many of which are extensions of the state pattern too. Domínguez et al. [11] compiled a table summarizing the features of many approaches, including those mentioned above.

3. Requirements on an alternative approach

The previous section explained the drawbacks of the traditional approaches. But how should an alternative look like?

What we are looking for, is an approach that is first of all fully object-oriented. All externally visible components shall be objects. That means, not only the state machine itself shall be an object, but also its interface to the outside world shall use objects. This leads to the postulation that also the events must be objects. It is not self-explanatory why we should still use enumerated numbers to