Reengineering C/C++ Source Code by
Transforming State Machines*

Roland Knor, Georg Trausmuth, and Johannes Weidl

Technical University of Vienna, Distributed Systems Group,
Argentinierstr. 8/184-1, A-1040 Vienna, Austria, Europe,
{knor, trausmuth, weidl}@infosys.tuwien.ac.at,
WWW home page: http://www.infosys.tuwien.ac.at

Abstract. State transition mechanisms are widely used in software engineering to implement state/event-dependent behavior. In C/C++, the implementation of state machines using 'switch/case' or 'if' statements causes problems in readability, understandability, maintainability, and modification. We have developed both a source code pattern searching tool capable of searching for state machine occurrences and a design pattern to replace state machines with C++ generic components. With 'ESPaRT' (Enhanced String Pattern Recognition Tool) we search for state machines in the source code. Examples of patterns for state machine detection are given in the paper. The state machine information is extracted and the state machine code is replaced by generic components following the 'generic Harel State Machine Engine' (HSME) design pattern.

1 Introduction

A common and widely used technique to implement the dynamic behavior of reactive systems—and especially embedded systems—is the use of state machines. Depending on the current system state and incoming events issued by a source in the environment, a well-defined action (sequence of program instructions) is carried out and a new system state is set. In automata theory, a finite state machine (FSM) is defined as a set of states, a set of events, an initial state, a set of final states, and a state transition function (for the formal definition see [10].) The state transition function determines for each valid pair (current state, incoming event) the system state to switch to. The visual counterpart of state machines—state transition diagrams—are used to model and simulate reactive systems.1

* This work is funded by the European Commission under ESPRIT framework IV contract no. 20477 'Architectural Reasoning for Embedded Systems (ARES)' and is pursued by Nokia RC Finland, ABB Norway, Philips RC Holland, Imperial College, Technical University Madrid, and Technical University of Vienna.

1 The new system state can be the same as the current.

2 Reactive systems are systems that have continually to react to events from their environment.

© Springer-Verlag Heidelberg Berlin 1998
As with nearly every concept, there are many ways to implement a state machine in C/C++. The probably most common approach is utilizing the `switch/case` construct\(^3\). A so-called `state variable` is used to store the current state. If an event arrives, the current state is determined (using a `switch/case` construct.) Then, the event is determined (using another `switch/case` statement nested in the first) and the appropriate program code is executed in the corresponding case block. Usually, the last statement in the case block manipulates the current system state.

We argue, that such an implementation has deficiencies. In general, there is no separation of the code-parts processing the state machine (the so-called `state machine engine`), the state transition information and the code to be executed (the `actions`.) Furthermore, there is no explicit representation of the system states, events, and the state transition information. This can prevent a reader from understanding the dynamic behavior modeled with the state machine, especially if the state/event space\(^4\) is large. Since readability and understandability are an essential basis for modification and enhancement of the state machine or for reverse engineering issues, we take effort to identify state machines in the source code and replace their implementation with an improved one.

Basically, such a venture has to offer three techniques. First, we must be able to identify state machines in the source code. We can look manually for state machine instances, use reverse engineering tools\(^5\), or apply pattern searching tools such as ‘grep’ or more sophisticated ones that, for instance, parse the code to generate and use an abstract syntax tree (see [5].) Second, we have to analyze the state machine and identify the state and event space, the state transition function, and the actions executed at state transitions. This, as well, can be done manually or using an appropriate tool supporting the software engineer in the analysis task. Third, we have to replace the state machine implementation with a more sophisticated one, which remedies the deficiencies stated above, using the information of step two.

For the first task we use our pattern searching tool ESPaRT. We present the concepts of ESPaRT in Section 2 and explain how it can be used to detect state machines in the source code. For the third task, we have developed a set of C++ generic components in the HSME design pattern. Section 3 gives a short survey on the HSME components. In Section 4, we discuss the transformation process in detail and give an outlook on an appropriate tool supporting the task. Section 5 surveys work related, in Section 6 we draw some conclusions.

## 2 Finding State Machines Using ESPaRT

The simplest way of finding state machines in the source code is a manual search using an editor. However, a tool that is supporting the search for occurrences of

\(^3\) Readers not familiar with C/C++ are referred to [7].

\(^4\) The state space is the set of all possible states, analogously the event space is the set of all possible events.

\(^5\) A comparison of reverse engineering tools can be found in [1].