

Hierarchical State Machines: a Fundamentally Important Way of Design



**Hierarchical State Machines:
a Fundamentally Important Way
of Software Design**

PARC Forum
May 6, 2004

Miro Samek
miro@quantum-leaps.com

Quantum Programming
Copyright © 2003 by Miro Samek. All Rights Reserved.


Objectives

- Describe reactive systems and the traditional approach to event-driven programming
- Introduce UML statecharts, a.k.a. Hierarchical State Machines (HSMs)
- Show how the basic concepts behind HSMs relate to OOP and quantum physics
- Show how a good implementation of HSMs leads to a **paradigm shift** in programming
- Questions are welcome at any time!

© Miro Samek • www.quantum-leaps.com • 2

Reactive Systems

- Almost all computer systems are **event-driven**
- After recognizing the event, they **react** by performing the appropriate computation
- The main programming challenge is to quickly pick and execute the **right** code in reaction to an event
- The problem is not at all trivial and leads to control inversion compared to the traditional data-processing systems




© Miro Samek • www.quantum-leaps.com • 3

Hierarchical State Machines: a Fundamentally Important Way of Design

The Traditional Event-Action Paradigm


- The currently dominating approach in programming reactive systems is the venerable **event-action paradigm**
- You drag-and-drop the graphical components and map events directly to snippets of **code**
- But then more and more `if`s and `else`s must be added to select the response based on the **context**, until no human being really has a good idea what part of the code gets executed in response to any given event ("spaghetti")



© Miro Samek • www.quantum-leaps.com • 4

Capturing the Context Explicitly

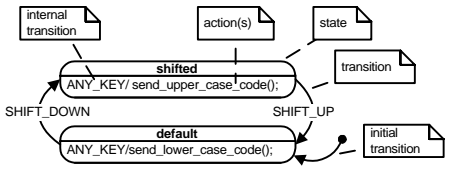
- An event alone doesn't determine the actions to be executed in response to that event. The current **context** is at least equally important
- Neglecting the context results in "spaghetti" code
- State machines** make the response to an event explicitly dependent on the context of the system
- State** captures the relevant context very efficiently by abstracting away all possible (but irrelevant) event sequences and capturing only the **relevant** ones



© Miro Samek • www.quantum-leaps.com • 5

State Diagrams

- State machines have a compelling and intuitive graphical representation in form of state diagrams
- State diagrams are directed graphs in which nodes denote states and connectors denote transitions
- The UML provides a standard notation and precise, rich semantics for state diagrams



© Miro Samek • www.quantum-leaps.com • 6

Hierarchical State Machines: a Fundamentally Important Way of Design

The Limitations of Traditional FSMs

- The traditional FSMs tend to become unmanageable, even for moderately involved reactive systems (“state explosion”)
- In practice, many states are similar, but classical FSMs have no means of capturing such commonalities and require repeating the same behavior in many states
- What’s missing in FSMs is a mechanism of factoring out the common behavior in order to **reuse** it across many states

© Miro Samek

• www.quantum-leaps.com •

7

Reuse of Behavior in Reactive Systems

- All reactive systems reuse behavior in a similar way
- For example, the characteristic look-and-feel of all GUIs results from the same basic pattern, which GUI programmers know as the “Ultimate Hook” [Petzold 96].
- A GUI application has the first shot at every event; thus, the application can customize every aspect of its behavior. All unhandled events flow back to the higher level (the GUI system), where they are processed according to the standard look-and-feel.
- This is an example of **programming-by-difference** because the application programmer needs to code only the differences from the standard system behavior.

© Miro Samek

• www.quantum-leaps.com •

8

Introducing Statecharts

- Harel **statecharts** bring the “Ultimate Hook” pattern to the logical conclusion by combining it with the state machine formalism.
- The UML state machines [OMG 01] are an object-based variant of Harel statecharts [Harel 87]. They incorporate several concepts similar to those defined in ROOMcharts, a variant of statechart defined in the ROOM modeling language [Selic+ 94].
- The most important innovation of statecharts is the introduction of *hierarchically nested states* (that’s why statecharts are also called **Hierarchical State Machines**).

© Miro Samek

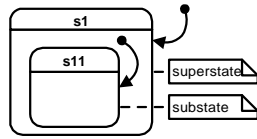
• www.quantum-leaps.com •

9

Hierarchical State Machines: a Fundamentally Important Way of Design

The Semantics of State Nesting

- If a system *is in* the nested state s11 (called **substate**), it also (implicitly) *is in* the surrounding state s1 (called **superstate**)
- Any event is first handled in the context of the substate s11, but all unhandled events are automatically passed over to the next level of nesting (s1 superstate)
- The substates need only define the **differences** from the superstates, and otherwise can easily share (**reuse**) behavior defined in higher levels of nesting

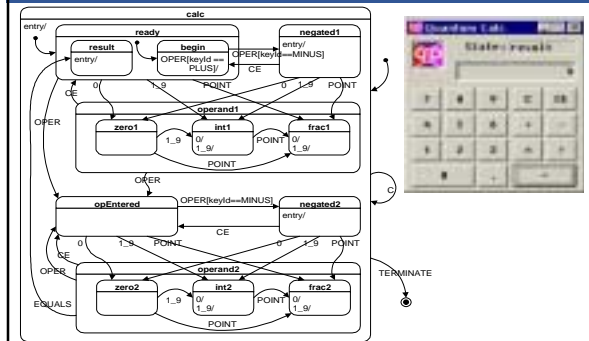


© Miro Samek

• www.quantum-leaps.com •

10

An Example: Calculator That Works



© Miro Samek

• www.quantum-leaps.com •

11

O-O Analogy – Behavioral Inheritance

- State nesting lets you define a new state rapidly in terms of an old one, by reusing the behavior from the superstate
- State nesting allows new states to be specified **by difference** rather than created from scratch each time
- State nesting lets you get new behavior almost for free, **reusing** most of what is common from the superstates
- The fundamental character of state nesting comes from the combination of hierarchy and programming-by-difference, which is otherwise known in software as **inheritance**
- State nesting leads to **behavioral inheritance** [Samek 00, 02]

© Miro Samek

• www.quantum-leaps.com •

12

Hierarchical State Machines: a Fundamentally Important Way of Design

O-O Analogy – Initialization and Cleanup

- HSMs allow states to have optional entry actions executed automatically upon the entry to the state and exit actions executed upon the exit (Moore automata)
- The value of entry and exit actions is that they provide means for guaranteed initialization and cleanup, much like class constructors and destructors in OOP
- Entry and exit actions are particularly useful in conjunction with the state nesting, because all exited/entered state contexts are properly initialized/cleaned-up
- The order of execution of entry actions must always proceed from the outermost state to the innermost state. The execution of exit actions proceeds in exact opposite order

© Miro Samek

• www.quantum-leaps.com •

13

O-O Analogy – LSP

- Liskov Substitution Principle (LSP) is a universal law of generalization. In the traditional formulation for classes LSP requires that a subclass can be freely substituted for its superclass
- Because behavioral inheritance is just a specific kind of inheritance, the LSP can (and should) be applicable to nested states as well as classes
- LSP generalized for states means that the behavior of a substate should be consistent with the superstate
- Compliance with the LSP (for states) allows you to build better (correct) state hierarchies that make efficient use of abstraction

© Miro Samek

• www.quantum-leaps.com •

14

O-O Analogy – Refactorings

- Because of the similarities between O-O concepts and the ideas underlying HSMs both state hierarchies and class taxonomies, at some point of their lifecycle, need restructuring to continue to evolve
- The same general *refactorings* are applicable both to O-O systems and to HSMs:
 - Refactoring to generalize – creating a common superclass (creating a common superstate)
 - Refactoring to specialize – deriving subclasses from a common base (nesting substates in a common superstate)

© Miro Samek

• www.quantum-leaps.com •

15

Hierarchical State Machines: a Fundamentally Important Way of Design

O-O Analogy – Design Patterns

“One thing expert designers know not to do is solve every problem from the first principles” [GoF 95].

- Experienced designers repeatedly realize good state machines, while new designers are overwhelmed by the options available and tend to fall back on convoluted ifs and elses and multitude of flags they have used before
- Experts recognize the similarities among problems and reuse proven solutions that work (design patterns)
- State patterns focus on effective ways of structuring states, events, and transitions.
- First state patterns catalogs: [Douglass 99, Samek 02]

© Miro Samek
www.quantum-leaps.com
16

Beyond Object Oriented Programming

- Most recent trends in programming (such as components, patterns, and frameworks) pertain to ways of combining many fine-granularity elements into systems
- None of these trends address the *internal* structure of a class.
- The traditional OO method, for example, stops short at the boundary of a class, leaving the *internal* implementation of individual class methods to mostly procedural techniques.
- Behavioral inheritance and the O-O analogy add another dimension to the OOP, because they allow many O-O methods to be extended and applied *inside* reactive classes.

© Miro Samek
www.quantum-leaps.com
17

Quantum Metaphor – States

- The significance of state machines goes beyond software. At the most *fundamental* level, all microscopic objects exhibit state behavior
- For example, the quantum states of the hydrogen atom are *naturally* hierarchical and comply with the LSP

$$H|nlm\rangle = \frac{-13.6eV}{n^2}|nlm\rangle, \quad n=1,2,3\dots$$

$$E_l^2|nlm\rangle = \hbar^2 l(l+1)|nlm\rangle, \quad l=0,1,2\dots n-1$$

$$L_z|nlm\rangle = \hbar m|nlm\rangle, \quad m=-l, -l+1, \dots, l$$

2
entry/n=2

2S
entry/l=0, m=0

2P
entry/l=1

2P-1
entry/m=-1

2P0
entry/m=0

2P+1
entry/m=+1

- State nesting expresses **symmetries** within the system.

© Miro Samek
www.quantum-leaps.com
18

Hierarchical State Machines: a Fundamentally Important Way of Design

Implementing State Machines – Overview

(a) Event/State Table: A table with columns for events (ev E1, ev E2, ev E3, ev E4) and rows for states (stateA, stateB). Arrows show transitions from stateA to stateB on ev E1, and from stateB to stateB on ev E2, ev E3, and ev E4.

(b) State Table: A table with columns for states (stateA, stateB, stateC) and rows for events (ev E1, ev E2, ev E3). Arrows show transitions from stateA to stateA on ev E1, from stateA to stateB on ev E2, and from stateA to stateC on ev E3.

(c) Handlers: A state (state) with three outgoing arrows to stateA handler, stateB handler, and stateC handler.

© Miro Samek • www.quantum-leaps.com • 19

Implementing HSM (C++)

```

OSTATE Cal c : : begin (OEvent const *e) {
    switch (e->sig) {
        case Q_ENTRY_SIG:
            dispatchState("begin");
            return 0; // event handled
        case OPER_SIG:
            if (((Cal cEvt *)e)->keyId == IDC_MINUS) {
                Q_TRAN(&Cal c : : negated1);
            }
            return 0; // event handled
    }
    return (OSTATE)&Cal c : : ready; // superstate
}
    
```

• Coding a non-trivial HSM turns out to be an exercise in following a few simple rules.

© Miro Samek • www.quantum-leaps.com • 20

Good HSM Implementation is Important...

- With just a bit of practice, you will forget that you are “translating” state models into code; rather, you will *directly* code state machines in C or C++, just as you directly code classes in C++ or Java.
- At this point, you will no longer struggle with convoluted if-else statements and multitude of flags. You will start *thinking* at a higher level of abstraction.
- Thus, a sufficiently small and truly practical implementation of HSMs can trigger a **paradigm shift** in your way of thinking about programming reactive systems. I call this paradigm Quantum Programming (QP) [Samek 02, QP 04].

© Miro Samek • www.quantum-leaps.com • 21

Hierarchical State Machines: a Fundamentally Important Way of Design

A Quantum Programming Language?

- From the inception, HSMs have been treated as purely visual formalism that requires visual design-automation tools (such as Harel's STATEMATE).
- However, programmers don't have to leave their favorite programming languages in order to work directly with higher level concepts, such as hierarchical states, events, and transitions.
- At this time, these concepts are just external add-ns (low-level design patterns) to C or C++.
- However, they lend themselves to being *natively supported* by a "quantum" programming language, in the same way that abstraction, inheritance, and polymorphism are natively supported by object-oriented programming languages.

© Miro Samek

• www.quantum-leaps.com •

22

An RTOS of the Future?

- The greatest demand for third-party software in the embedded industry is for the real-time operating system.
- Concurrently executing state machines asynchronously exchanging events provide a better and safer alternative to conventional RTOS based on mutual exclusion and blocking.
- Sufficiently robust *infrastructure* for executing state machines (event processor, execution threads, event queuing, event dispatching, and timing services) can be reused across many projects rather than being invented each time
- Most of commercial design-automation tools are build around such *frameworks*
- A RTOS of the future could be a **state machine framework**

© Miro Samek

• www.quantum-leaps.com •

23

References

- [Gamma+ 95] Gamma, Erich, et al., *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [Harel 87] Harel, David, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, 8, 1987, pp. 231-274.
- [OMG 01] Object Management Group, Inc., *OMG Unified Modeling Language Specification v1.4*, <http://www.omg.org>, September 2001.
- [QP 04] Quantum Programming website at www.quantum-leaps.com
- [Samek+ 00] Samek, Miro and Paul Y. Montgomery, "State-Oriented Programming", *Embedded Systems Programming*, August 2000 pp. 22-43
- [Samek 02] Samek, Miro, *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*, CMP Books, 2002
- [Selic+ 94] Selic, Bran, Garth Gullekson, and Paul. T. Ward, *Real-Time Object Oriented Modeling*, John Wiley & Sons, 1994.

© Miro Samek

• www.quantum-leaps.com •

24
