Raising the source code abstraction level by using generic components for state machines

Johannes Weidl
Esprit IV #20477 "ARES" research project

Information Systems Institute, Distributed Systems Department
Technical University of Vienna
Argentinierstr. 8 / 184-1, A-1040 Vienna, Austria
+43-1-58801-4410, J.Weidl@infosys.tuwien.ac.at

Abstract

Various software architectures use state transition mechanisms as a major building block. As an example, finite state machines (FSMs) and their graphical counterpart—state transition diagrams—are heavily used e.g. for the specification of various kinds of protocols such as network protocols (TCP/IP) or protocols for infrared data transmission (IrDA). Many embedded systems, e.g. telephone switching systems and television control, are based directly on state machines. Introducing generic components for state machines can raise the source code abstraction level from 'hard-coded' control flow decisions based upon 'switch/case' and 'if' statements to a more flexible implementation model of control flow. This paper shows that it is possible to use statecharts and their advanced mechanisms from specification to implementation of reactive systems by simultaneously raising the source code abstraction level by object-oriented and table-based means. Statecharts developed by David Harel [Harel87] extend state transition diagrams with a notion of hierarchy and concurrency thus bringing major simplification when system complexity increases. Furthermore, the table-based approach shows that a certain degree of genericity and thus flexibility can be reached without major losses in efficiency during the design and implementation of generic components.

1 Introduction

In the ARES project we look at both the forward and the reverse direction in the software engineering process. The forward direction leads from the requirements using intermediate design representations of different abstraction level (e.g. data-flow diagrams, structure charts, etc.) to the implementation. The reverse direction tries to extract these intermediate representations from the source code using domain knowledge and to restore the original design of a software system.

Embedded systems, which is the application area of our project, are often based on state machines to implement their dynamic behavior. We are, therefore, interested in the identification of state machines in the source code and their representation at a higher level of abstraction. We have focused on the creation of generic components for state machines to raise the source code abstraction level and thus achieve an intermediate representation of a higher abstraction level.

To implement the dynamic behavior of reactive[1] systems, programmers often 'hard-code' finite state machines[2] (FSMs) using large, multipart conditional statements such as the C++ statements *switch/case* and *if*. Such an implementation technique usually maps the high-level concepts 'state' and 'event' to integral data types (e.g. int, char, etc.). The state transition function—yielding a successor state for each pair of possible system state and event—is embedded in a possibly huge conditional statement, deciding on a next state in dependence on the current system state and the next[3] event. Because of this mapping to low-level programming language entities I call the source code abstraction level of such a state machine implementation *low*. Thus, raising the source code abstraction level means mapping the concepts of state, event, and state transition function to higher-level concepts of the appropriate programming language. This decreases the effort for understanding the control flow of a specific system implementation, e.g. for purposes of program/system understanding, modification, and reuse.

However, the major benefit of a low-level *statement-based* approach is its efficiency in terms of execution time and lines of code. The low execution time and small number of code lines justify the approach especially for use in embedded systems which are usually subject to sparse resources. Consequently, an alternative approach such as the presented that is supposed to be used in practice has to meet certain efficiency requirements, i.e. it should be about as efficient[4] as conventional solutions.

The approach presented in this paper maps the concept of a state machine (which additionally is able to process statechart specifications (see section 2) rather than simple state transition diagram specifications) to *generic* components. Component programming is a software engineering approach that is based on using standard software components in the software engineering process to advance from line-by-line development to component based development [Jazayeri95, NGT92]. Thus, a component can be seen as a high-level programming language concept. Furthermore, *generic* components [MS89, MS94] are designed to be generally applicable and usable in many contexts and thus their use further increases the source code abstraction level by reducing the number of possible mappings from 'real-world concept' or 'domain concept' to 'programming language concept' or simply 'set of source code statements'.

A design pattern is used to solve recurring problems in the software development process [GHJV95]. I present a pattern called 'Extended State' that integrates several software development approaches, including David Harel's visual formalism of statecharts to specify dynamic system behavior [HG96**,** Harel87**,** Harel88], Alexander Ran's techniques to model states as classes [Ran94**,** Ran96], and work on generic state machine *engines*. The suggested design pattern introduced in section 2 and evaluated in section 3 will allow the use of statecharts for graphical specification of reactive system behavior and provide the advanced statechart features down to the state machine implementation level. As important as the homogenous application of statechart features during the specification, design, and implementation process is the issue of providing a

[1] Reactive systems are characterized by being event driven, continuously having to react to external and internal stimuli [HP85].

[2] For a mathematical definition of FSMs see [HU79].

[3] The term 'next' does not necessarily imply a chronological order in terms of system time, usually the events are stored in an event queue which itself orders the incoming events according to specific criteria.

[4] 'About as efficient' means that efficiencies should not differ by magnitudes.

state machine engine [RBPEL91], which processes the high-level state transition information and thus implements the correct system behavior.

2 The 'Extended State' Design Pattern

The 'Extended State' design pattern is expected to simplify specification, design and implementation of reactive systems. For the specification and design of dynamic system behavior the graphical specification technique of statecharts is used.

Statecharts are based on the conventional state transition diagrams and additionally provide the mechanisms *depth*, *orthogonality* and *broadcast communication* [Harel87]. Using statecharts, several states can be grouped as substates inside a superstate and, thus, a notion of depth or modularization is achieved. Since the number of states in a linearly growing system grows exponentially [Harel88], statecharts are a powerful tool in specifying the dynamic behavior of reactive systems. In a single superstate more than one substate can be concurrently active. This is referred to as *orthogonality of states*. Directed arcs between states are labeled with events meaning that the occurrence of a particular event causes a state transition from the source to the destination state of the corresponding arc. State transitions can be blocked by conditions so that a transition is only carried out when the condition evaluates to true. When a state transition takes place it is possible to generate internal events. Such as external events that cause state transitions in all orthogonal components to which they are relevant using a broadcast mechanism, internal events are reported to all components inside the statechart possibly leading to further state transitions. Statecharts are not only a tool for visualization but are based on a clear mathematical formalism ([Harel88]) and, hence, the graphical concept of statecharts is called *visual formalism*.

The 'Extended State' design pattern consists of two independent approaches: The first approach (section 2.1) extends an existing object-oriented design pattern called 'State' to enable the use of the statechart features. The 'State' pattern is based on techniques to model states as classes and its description can be found in [GHJV95]. The second approach (section 2.2) is table-based, i.e. the transition information is defined in a table and a generic state machine engine processes this table.

2.1 The 'Harel State' pattern (abbr. HS)

[Ran94] introduces techniques to model states as classes, because commonly used object-oriented languages do not offer sufficient support for modeling states and state-dependent behavior. Especially state-polymorphism, i.e. the implicit method selection based on an object's state, is not supported. Ran suggests to model different *abstract* states (i.e. certain cases of behavior) as state classes (one state class for each abstract state) which define the state-dependent behavior in their methods. The state classes are grouped into *clusters* and represented to clients by a single class - the cluster *deputy*. The mechanism of *state-dependent dispatch* allows instances of the cluster deputy to exhibit state-dependent behavior by transparently calling the method of the state class that represents the current abstract state. 'Harel State' extends 'State' to cope with the statechart mechanisms depth, orthogonality, and broadcast communication so a system modeled with statecharts can be more directly transferred into code.

Using this design pattern, the dynamic behavior of the system that is to be implemented has to be specified according to the system requirements using a statechart. From the resulting statechart a so-called *state diagram* is derived. The state diagram shows the hierarchy of states beginning with the global superstate and relating all substates according to the state-of relation [Ran94]. The state diagram then is interpreted as a subclass (inheritance) hierarchy of state classes which is implemented accordingly. Events are modeled as methods in the state classes. When an external event arrives, a client has to call the deputy method that corresponds to the event. The deputy then calls the method of the state class that represents the current abstract state.

The statechart feature of depth is inherently given when using the state class approach because of the subclass relation. When a certain transition functionality is defined for a state class representing a superstate, the transition behavior is inherited to all subclasses of the state class (substates) and thus has only to be defined once for the superstate. The orthogonality feature is implemented by extending the original single state pointer—pointing to the state class representing the current abstract state—to an array of pointers. Alternatively, a list of state pointers could be used. To implement the broadcast communication feature on the arrival of an external or internal event, a loop over all state pointers is carried out to report the event to all currently active states. Internal events can be broadcast over the system by calling the corresponding deputy member function directly out of a state class method. Additional coding effort is required when the next system state is a superstate, usually consisting of more than one substate. In this case, it has to be determined how much and what substates get current states in order to add their state pointers to the current-state data structure. This has to be done recursively for all substates of the superstate until there is no superstate left.

2.2 The 'Generic Harel State Machine' pattern (abbr. HSME)

The HSME approach is a table-based approach, i.e. the transition information and the actions to be executed at state transitions are kept in a table. HSME uses an explicit state machine engine to process the data given in the state transition table and thus implement the specified system behavior, hence the specific name. States are not represented as classes but as C++ template *struct*s [Stroustrup93]. This approach uses templates to make the implementation independent of concrete types. The transition function is a C++ template struct and is fully instantiated at compile time. As much data as possible is kept static so that a maximum of the state machine code can be instantiated at compile time and the run-time part remains minimal. The only dynamic part is the state machine engine loop. The 'minimality' of the approach tries to address the efficiency requirements which are essential when it comes to the implementation of e.g. embedded systems.

A problem in this approach is the implementation of the depth or modularization feature. In HS, a certain transition functionality has to be defined only once in a state class to be inherited to all subclasses. The static table as used in this approach does not allow such a simplification, the next state and the appropriate action have to be specified in the table for all substates of a superstate. This is a conceptual problem rather than an implementational problem because a two-dimensional table as a flat model does not inherently provide means of modularization, depth or hierarchy. To implement the orthogonality feature the current implementation of HSME uses a state pointer array to store all active states. Broadcast communication is achieved by a loop over the state pointer array to broadcast internal and external events. When internal events are issued they are added to a queue in the state machine component and are processed ahead of all following external events.

To use a HSME state machine in a user's program first the states, events, and actions have to be defined. Then, the state transition table is specified using the already defined data structures and functions. At last the state machine template structure is instantiated, the initial state has to be defined, and the `execute` member function of the state machine component has to be called to start the processing.

3 Evaluation

3.1 The HS pattern

HS is based on the existing object-oriented design pattern 'State' and is extended in a way that state machine specifications derived from statecharts can easily and directly be implemented. The hierarchy of states is implemented as a class hierarchy using inheritance, events are modeled as methods of a deputy class representing this state cluster. The source code abstraction level is raised by the correspondence of events to methods, states to classes, and the state machine concept to a cluster of state classes represented by a deputy class. But although the state cluster represents a software component, no dedicated concept is used to make the component generic. A completely separated engine encapsulated in a single component does not exist. For this, the implementation of a new state machine means to rewrite or at least copy the state machine engine code. Since HS defines a state class for each abstract state, the HS implementation contains more lines of code compared to a table-based or a low-level approach. These shortcomings are compensated by a clear object-oriented approach explicitly modeling states and state transitions and providing the powerful mechanism of modularization.

3.2 The HSME pattern

The HSME pattern is based on a generic state machine engine as a separate software component working on table-based transition information. Using a table-based approach, transition information (equivalent to control flow or dynamic system behavior) can be modified and maintained conveniently by modifying table entries rather than source code statements. The source code abstraction level is raised by mapping the transition information to a static state transition table and by using an independent state machine engine component to process the table. The approach is generic in having separate software components which additionally are templates to be independent of the concrete types of data structures associated. Since the state machine engine and the state machine specification (state, event, transition function data structures) are independent software components, the same physical state machine engine implementation can execute all valid state machine descriptions provided by the user, the state transition information can be modified without going into low-level engine code, and the execution loop can be modified without unwillingly modifying state transition information (and vice versa). Furthermore, to change the engine functionality, in contrast to HS the engine component code has to be modified only once and then can be recompiled with all state-machine specifications concerned.

The 'minimality' of the approach (compile time code instantiation and minimal run-time part) preserves efficiency in terms of execution time and code size compared to low-level approaches.

4 Summary and Conclusion

This paper showed two possible implementations of generic components for state machines, focusing on the component programming paradigm, genericity and efficiency, Harel's visual formalism of statecharts, and Ran's techniques to model states as classes. A major goal is the simplification of specification, design, and implementation of reactive systems. The HSME design pattern—as a generic component approach—is expected to be significantly more flexible than a low-level coded state machine, but at the same time working about as efficient as the low-level approach and consuming about the same space. In striving for a more flexible modeling approach of reactive systems by using generic components, the HSME pattern can help to improve productivity, quality, and reuse of reactive system design, preserving efficiency and code size of low-level approaches.

5 References

[CS95] Coplien J. O. and Schmidt D. C., "Pattern Languages of Program Design", Addison-Wesley, 1995

[GHJV95] Gamma E., Helm R., Johnson R., and Vlissides J., "Design Patterns", Addison-Wesley, 1995

[Harel87] Harel D., "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, North Holland, 8(3), pp.231-274, 1987

[Harel88] Harel D., "On visual formalisms", *Comm. ACM*, 31, pp.514-539, 1988

[HG96] Harel D. and Gery E., "Executable Object Modeling with Statecharts", *Proceedings of ICSE-18*, IEEE, 1996

[HP85] Harel D. and Pnueli A., "On the development of reactive systems", *Logics and Models of Concurrent Systems*, NATO, ASI Series, vol. 13, K.R. Apt, Ed. Springer-Verlag, New York, pp.477-498, 1985

[HU79] Hopcroft and Ullman, "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley, 1979

[Jazayeri95] Jazayeri M., "Component Programming - a fresh look at software components", Fifth European Software Engineering Conference (Barcelona, September 25-28), 1995

[MS89] Musser D. R., and Stepanov A. A., "Generic programming", *ISSAC'88 Symbolic and Algebraic Computation Proceedings*, published as P. Gianni, editor, *Lecture Notes in Computer Science*, 358, Springer-Verlag, 1989

[MS94] Musser D. R. and Stepanov A. A., "Algorithm-oriented generic libraries", *Software - Practice and Experience*, 24(7), pp.623-642, July 1994

[NGT92] Niederstrasz O., Gibbs S., and Tsichritzis D., "Component-oriented software development", *Communications of the ACM*, Association for Computer Machinery, 33(9), pp.160-165, September 1992

[Ran94] Ran A., "Modelling States as Classes", Nokia Research Center, Software Technology Laboratory, P. O. Box 45, 00211 Helsinki, Finland, 1994

[Ran96] Ran A., "MOODS - Models for Object-Oriented Design of State", Early version of Chapter 8 from [CS95]

[RBPEL91] Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorensen W., "Object-oriented modeling and design", Prentice Hall, Englewoods Cliffs, New Jersey, 1991

[Stroustrup93] Stroustrup B., "The C++ programming language", 2nd edition, Addison Wesley, Jun. 1993