1 Introduction

Most of the current object oriented methodologies, such as OMT [1], Booch [2] method, Use-case method [3] and Unified Modeling Language (UML) [4], suggest to represent the dynamic behavior of an object oriented system by a set of state transition diagrams. These methodologies usually use Harel's statecharts [5, 6, 7] which offer a number of extensions, such as state hierarchy and concurrent states, to the usual state transition diagrams.

A statechart represents the complete behavior of a single class of objects. To create statecharts, the object oriented methodologies suggest to first create some sort of scenarios represented by event trace diagrams [1] or message sequence charts [4] which show the interactions between objects while the system is running. However, the methodologies do not explain how statecharts can be built from a set of scenarios.

In this paper, we present a set of rules to create statechart for a particular class of objects from a set of event trace diagrams in which the objects of the class participate. First a simple statechart is created and then rules are used to simplify and compact the statechart. Our method is iterative. New event traces can be added to statecharts created earlier from some other event traces.

2 Event Traces and Statecharts

An event trace diagram represents a scenario, which is a sequence of events that occurs during one particular execution of a system. The event trace diagram shows each object as a vertical line and each event as a horizontal arrow from the sender object to the receiver object. Time increases from top to bottom.

A state is an abstraction of the attribute values of an object. A state specifies the response of the object to input events. The response of an object to an event may include an action and/or a change of state by the object. A state corresponds to the interval between two events received by an object. Events represents points in time; states represents intervals of time.

A statechart relates events and states. When an event is received, the next state depends on the current state as well as the event. A change of state caused by an event is called a transition. A statechart is a graph whose nodes are states and whose directed arcs are transitions labeled by event names and possibly action names. The statechart specifies the state sequence caused by an event sequence. If an object is in a state and an event labeling one of its transitions occurs, the object enters the state on the target end of the transition. The transition is said to fire.

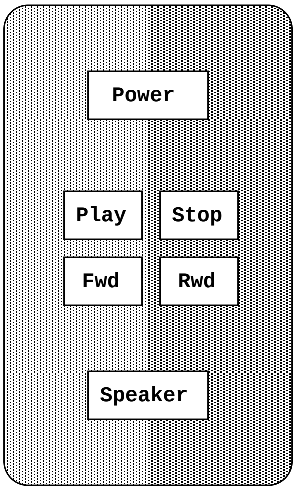

Figure 1: Remote control device for the cassette player

3 Creating Simple Statecharts

We use a simple Cassette Player system throughout the paper to demonstrate our approach. The player is operated with a remote control device having several buttons (Figure 1). We suppose there are two classes in the system: Device, which represents the remote control device, and Controller, which implements the actual behavior of the player when various buttons are pressed. Whenever some button is pressed, the Device informs the Controller, which in response performs some action and changes the state of the player. We use our approach to construct statechart for the Controller class from a set of event trace diagrams.

A statechart describes the behavior of a single class of objects. The sequence of events in an event trace diagram corresponds to paths through the statecharts of the corresponding objects. A statechart of a class of objects should have paths corresponding to all event trace diagrams in which the objects of that class participate. Therefore, to construct a statechart for a class of objects, we should consider the vertical lines that correspond to the objects of that class in all the event traces.

For an object in the event trace diagram, the incoming arrows designate events received by the object and the outgoing arrows designate actions performed by the object. The interval between any two consecutive events specifies a possible state. If there is an action between two events, the action is executed while entering the new state.

We believe that statecharts should be built in phases. In the first phase, the following three rules are applied to each of the event traces in which the objects of the class participate:

RULE 1: Before receiving any event, the object is in the default state.

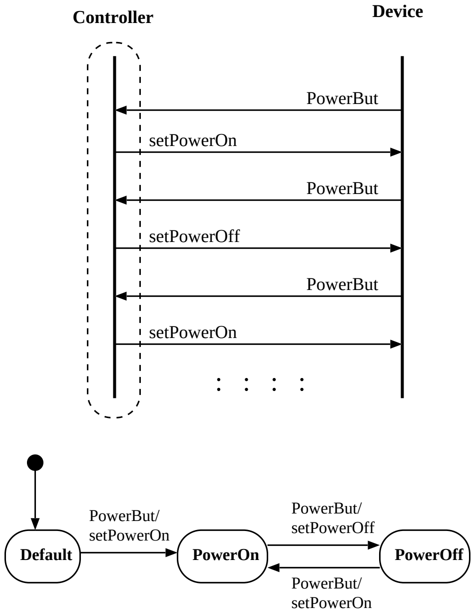

Figure 2: Event trace 1 and the corresponding statechart 1

RULE 2: Incoming arrows are events; they become transitions.

RULE 3: Intervals between events become states. Outgoing arrows at the intervals are actions; they become actions of the transitions leading to the states.

Figures 2 and 3 show two event traces and the corresponding statecharts built following the above rules. Each transition has a label, which consists of the event name followed by a slash ("/") followed by the action name. Each state is given a unique name. Any name can be given to a state, but it makes sense to derive the name from the action of the transition leading to the state, as we did. Next, the following rule is used to obtain a single statechart.

RULE 4: A single *main* statechart for a class is obtained by combining all the *partial* statecharts created from different event traces. The default states in the partial statecharts represent a single default state in the main statechart. States and transitions from all the partial statecharts are added to the main statechart. If a transition or state is common in the partial statecharts, it is taken only once in the main statechart.

Suppose we also have the statecharts of Figures 5 and 4 from some other event trace diagrams. Figure 6 shows the statechart obtained by combining the statecharts of Figures 2, 3, 4 and 5.

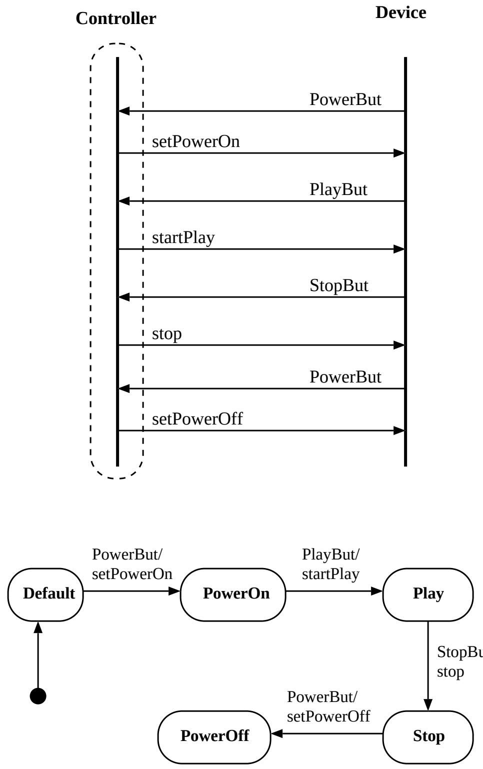

Figure 3: Event trace 2 and the corresponding statechart 2

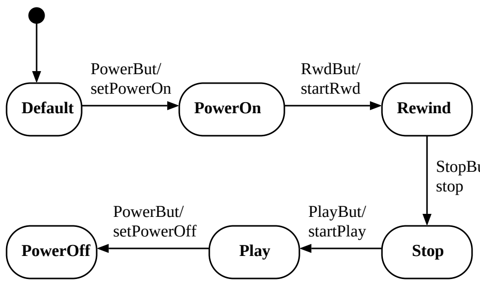

Figure 4: Statechart 3 obtained from some event trace diagram

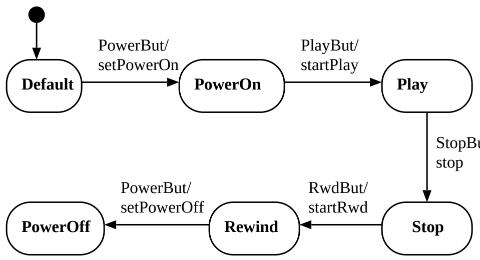

Figure 5: Statechart 4 obtained from some event trace diagram

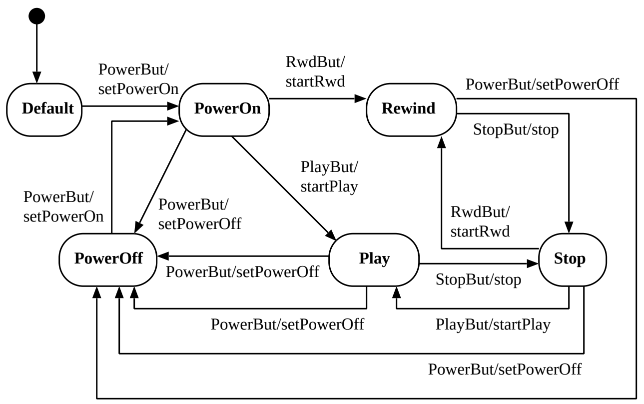

Figure 6: The main statechart obtained by combining the partial statecharts

4 Deleting unnecessary states

After creating a simple statechart for a class, it can be simplified by deleting unnecessary states and transitions. A statechart with lesser number of states and transitions is easier to comprehend. Before presenting the rule for deleting unnecessary states, we define two functions for a state: S_{out}, the set of all out-going transitions from state S; and S_{in}, the set of all in-coming transitions to state S. The two functions can be mathematically defined as:

$S_{out} = \{(e, a, st)$, such that when the current state is S and event e occurs, action a is executed and the state is changed to $st\}$

$S_{in} = \{(e, a, sf)$, such that when the current state is sf and event e occurs, action a is executed and the state is changed to $S\}$

RULE 5: If $S1_{out} \subseteq S2_{out}$, then $S1$ and $S2$ can be replaced with a single state (say $S3$), such that $S3_{out} = S2_{out}$ and $S3_{in} = S1_{in} \cup S2_{in}$.

Using the above rule, the Default and PowerOff states in Figure 6 can be combined to form a single state (PowerOff). Similarly, the PowerOn and Stop states can be combined together to form a single state (Stop). The result is shown in Figure 7.

5 State Hierarchy

The concept of state hierarchy can be used to decrease the number of transitions in a statechart. A transition from a superstate is inherited by each of the substates.

RULE 6: If $S1, S2, .., Sn$ have the same transition (e, a, s), then a superstate (say S) having $S1, S2, .., Sn$ as its substates can be introduced such that the transition (e, a, s) will be removed from each of the substates $(S1, S2, .., Sn)$ and added to the superstate S.

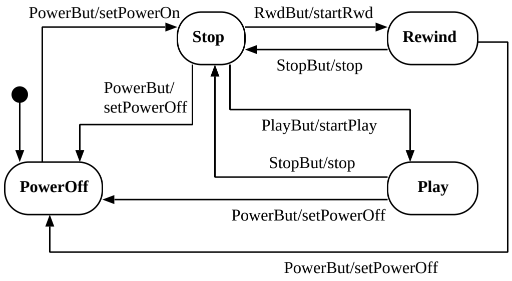

Figure 7: The simplified statechart with its states combined

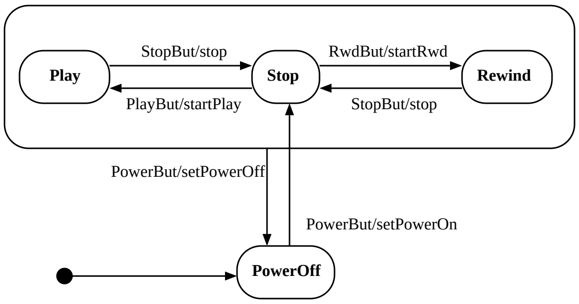

Figure 8: Statechart with state hierarchy

In Figure 7, there is a common transition (e = PowerBut, a = setPowerOff, s = PowerOff) from states Stop, Play and Rewind. Applying the above rule, we have the statechart of Figure 8.

When there is a state hierarchy, there is usually a default substate inside each of the superstate. When a transition leading to the superstate is executed, the default substate gets activated. We can use the following rule to divert a transition from a substate to the superstate.

RULE 7: Transitions towards a substate can be directed to its superstate if the substate is the default one.

Figure 9 shows the result when the above rule is applied to the statechart of Figure 8. We give the name PowerOn to the superstate because of the transition from the PowerOff state.

6 Concurrent states

It is not necessary to apply the statechart simplification rules to all of the event trace diagrams at once. Instead, a statechart can be constructed by considering only few

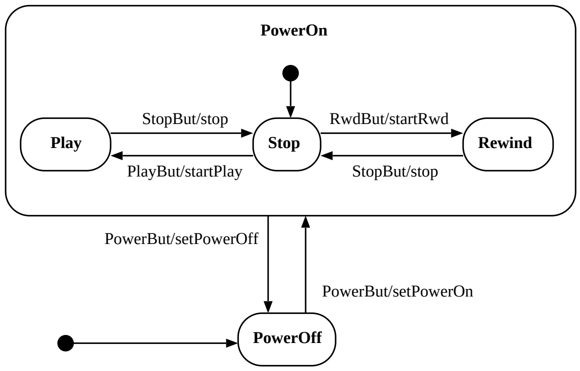

Figure 9: Statechart with a default substate inside a superstate

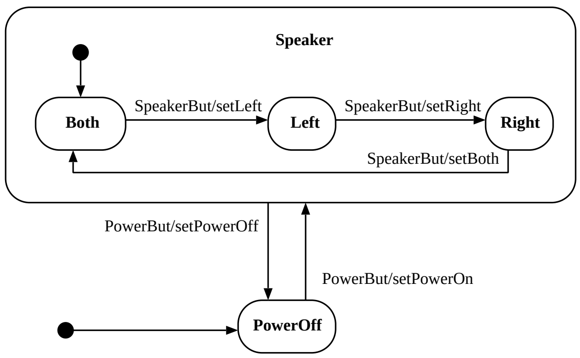

Figure 10: Statechart created from a different set of event trace diagrams

of the event traces and then the statechart can be extended as new event traces become available. We can even create a statechart from a set of event traces and then combine it with a statechart created earlier from a different set of event traces. The above rules can be used to combine the statecharts. This later combination of two or more independently created statecharts becomes very valuable if the underlying class of objects has some sort of concurrency. Concurrent states become active simultaneously whenever their superstate becomes active. This results in a very compact description of a complex system. Concurrent substates are also called AND-type substates, whereas the usual substates are called OR-type substates. Suppose that we have another statechart (Figure 10) for the Controller class of the player system created from another set of event trace diagrams. Now the following rule states that we have to introduce concurrent states in order to combine the statechart of Figures 9 and 10.

RULE 8: If there are two superstates $S1$ and $S2$, such that $S1_{in} = S2_{in}$ and $S1_{out} = S2_{out}$, then they must be combined with AND.

Using the above rule, we arrive at the statechart shown in Figure 11. We give the names Speaker and Player to

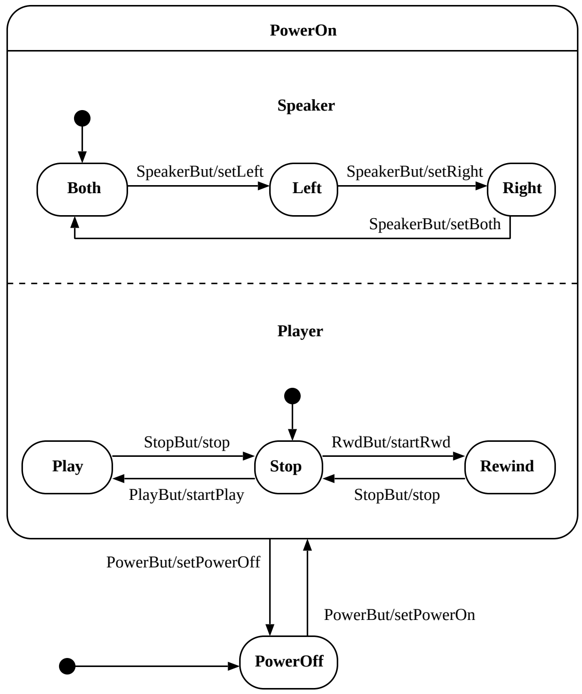

Figure 11: Statechart having concurrent states

the two AND-substates and the name PowerOn to the superstate of the substates.

7 Refining the Statechart

The major difference between an event trace diagram and a statechart is that an event trace diagram shows the sequence of events that occurs during one particular execution of a system whereas a statechart is supposed to represent the complete behavior of a class of objects. Event trace diagrams do not completely describe the dynamic model of a system. Therefore, even after considering all the event trace diagrams of a system, it is quite possible that the resulting statecharts are incomplete. We believe that there should be a refinement phase where the missing parts of a statechart are added to it.

In the cassette player remote control device, there is also a Fwd button which can be pressed and the system is supposed to respond to it in some way. The statechart of Figure 11 does not specify any behavior for a FwdBut event. In this particular example, we can guess that the behavior of the FwdBut event resembles with that of RwdBut and can refine the statechart as shown in Figure 12. However, sometimes we may not be able to guess the behavior and may need to create an extra event trace diagram to make the statechart completed. Applying RULE 6 about state hierarchy to Figure 12, we get the statechart of Figure 13.

The following rule can be applied to the final statechart to avoid the repetition of action names on transi-

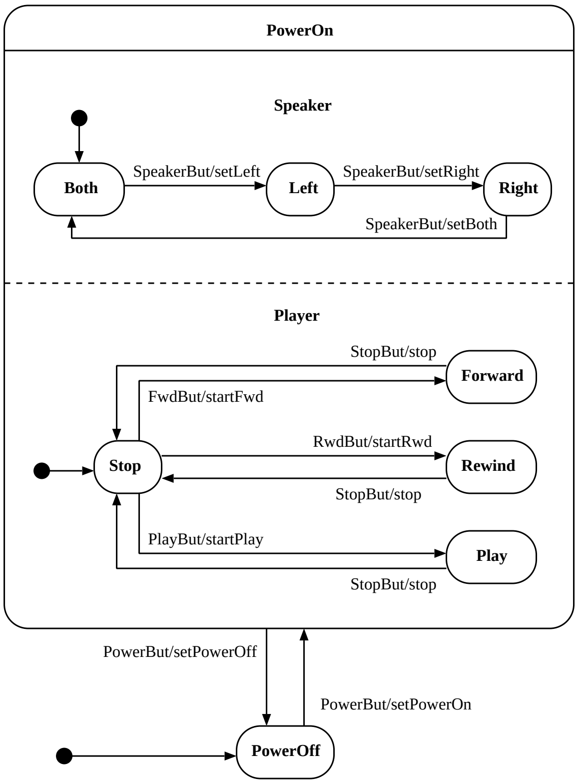

Figure 12: Statechart after adding a new state Forward

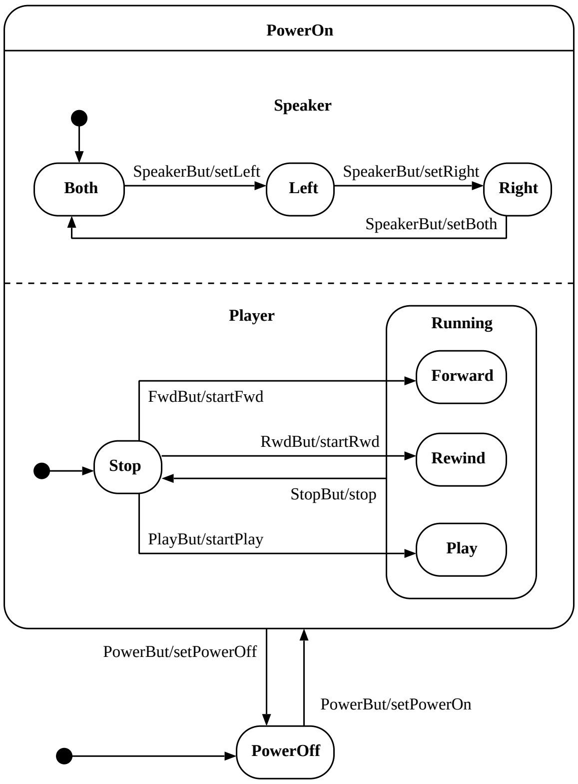

Figure 13: Statechart with further state hierarchy

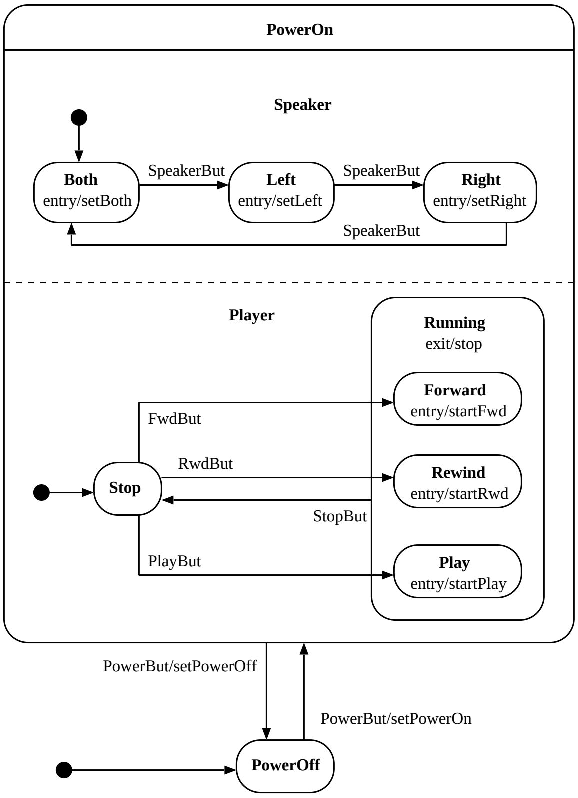

Figure 14: Statechart with *entry* and *exit* actions

tions.

RULE 9: If all the in-coming transitions of a state have the same action, the action can be made as the *entry* action of the state and removed from the transitions. Similarly, if all the out-going transitions have the same action, the action can be made the *exit* action and removed from the transitions.

Figure 14 shows the result when the above rule is applied to the statechart of Figure 13.

8 Related Work

The most related work is that of Koskimies and Makinen [8] who developed an algorithm which converts a sequence of event trace diagrams into state machines. The basis of their algorithm is the same as our RULES No. 2 and 3. The result of the algorithm is simple state machines without any state hierarchy and concurrent states. Whereas our set of rules produce real statecharts with almost all of the extensions described by Harel [5, 6, 7].

9 Conclusions

A set of rules have been developed through which statecharts can be created from event trace diagrams. First, basic rules are applied to build simple statecharts. Then simplification rules are applied to make the statecharts simple and compact by deleting unnecessary states and introducing state hierarchy and concurrency. In future work, we plan to implement our rules in a system which automatically converts a set of event trace diagrams into statecharts.

References

[1] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Eaglewood Cliffs, New Jersey, 1991.

[2] Grady Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, Redwood, California, 1991.

[3] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley, Reading, Massachusetts, 1992.

[4] Rational Software Corporation. *Unified Modeling Language (UML).* http://www.rational.com.

[5] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, (8):231–274, August 1987.

[6] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[7] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[8] Kai Koskimies and Erkki Makinen. Automatic synthesis of state machines from trace diagrams. *Software – Practice and Experience*, 24(7):643–658, July 1994.