Code Generation from Extended Finite State Machines

Insup Lee
Department of Computer and Information Science
University of Pennsylvania

Originally prepared by Shaohui Wang
Modified by Insup Lee for CIS 541, Spring 2010

Outline

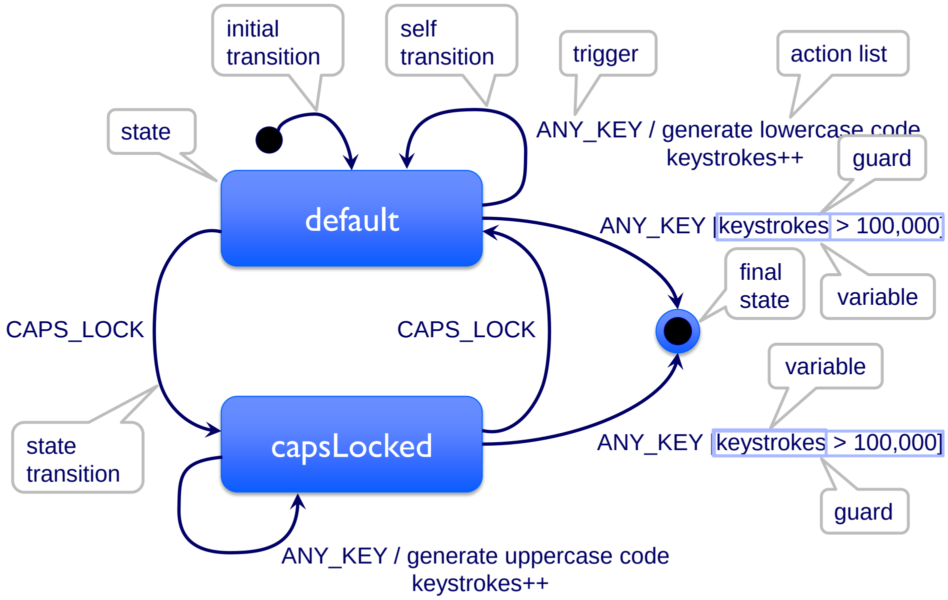
- Definition of Extended Finite State Machines
- General Code Generation Schemes
- Introduction to The EFSM Toolset

EXTENDED FINITE STATE MACHINES

Extended Finite State Machines

- A Keyboard Example
- Formal Definitions
- Variants

EFSM: A Keyboard Example



EFSM: Definitions

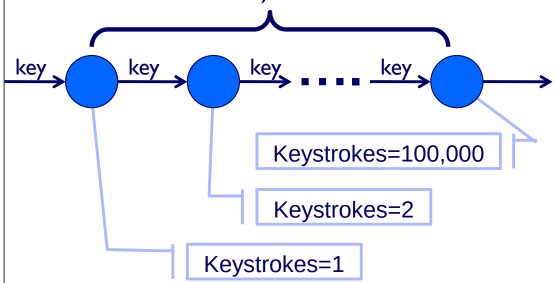
- An EFSM consists of
  - Extended States, Guards, Events, Actions, and Transitions.
- A **state** is a situation or condition of a system during which
  - some (usually implicit) invariant holds,
  - the system performs some activity, or
  - the system waits for some external event.
- An **extended state** is a state with "memory"
  - E.g., using 100,000 states to count 100,000 key strokes; or, using one extended state with a variable "count".

States vs Extended States

Example: Counting Number of Keystrokes (<= 100,000)

States

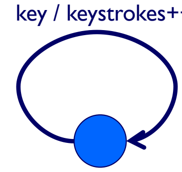
- One state represents one number
- Needs 100,000 states



No concept of variable 'keystrokes', using different states to represent them.

Extended States

- One state represents all
- Needs one state with one variable



One state and one variable to represent all.

EFSM: Definitions

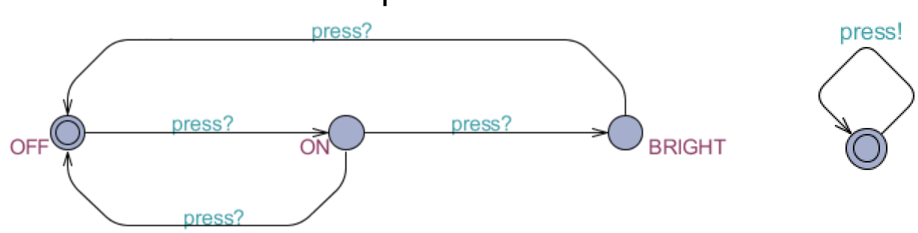
- **Guards** are
  - boolean conditions on extended state variables which enables or disables certain operations (e.g., change of states).
  - evaluated dynamically based on the extended state variable values.
  - immediate consequence of using extended state variables.
- An **event**
  - is an occurrence in time and space that has significance to the system.
  - may also be parametric which conveys quantitative info.

EFSM: Definitions

- **Actions**
  - are performed when an event instance is dispatched.
  - include
    - changing a variable;
    - performing I/O;
    - invoking a function;
    - generating another event instance; or
    - changing to another state (state transition).
- **Transitions**
  - can be triggered by events.
  - can also have a guard.

EFSM Variants: An Example

- Two Communicating EFSMs
  - with event channel "press"



- How to model "pressing twice fast"?
  - need time variables – timed automata

EFSM: Variants

- Input/output Variables
  - can serve as enhanced messages in communicating EFSMs
- Communicating EFSMs
  - input/output events are properly defined to convey info
  - EFSMs communicate by sending and receiving events via channels
- Timed Automata
  - "clock" variables are added, increasing automatically as
    - clock ticks (discrete) or
    - time elapses (continuous)
- Hierarchical EFSMs
  - inside a state, the system behavior is also like an EFSM

STATE MACHINE CODING SCHEMES

State Machine Coding Schemes

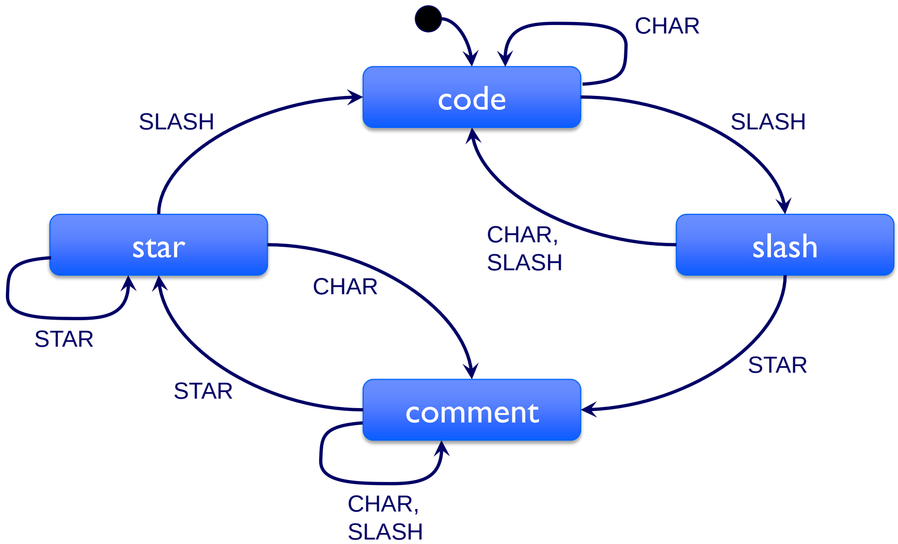
- State Machine Interface
- A Running Example
- Coding Schemes
  - Nested Switch Statement
  - State Table
  - Object-oriented State Design Pattern
  - Multiple-threaded Implementation
- Tradeoffs between EFSM Implementations

State Machine Interface

- Three methods
  - init() – take a top-level initial transition
  - dispatch() – dispatch an event to the state machine
  - tran() – take an arbitrary state transition
- Each coding scheme virtually implements these
- To use an EFSM in the main logic of the code
  - create an EFSM instance
  - invoke init() once
  - call dispatch()
    - repetitively in a loop
    - or sporadically based on detected events
  - dispatch() will then call the corresponding trans() function

The C Comment Parser Example

- A comment in the C language: /* comments */



The Nested `switch` Statement

- Nest switch statement with
  - a scalar state variable in the first level for states,
  - an event signal in the second level, and
  - transition logic (actions) in the innermost level

- Two alternatives
  - Switch with events first and then states
  - Use one switch that combines both.

The Nested `switch` Statement: Example

```
enum Signal {                               // enumeration for CParser signals
   CHAR_SIG, STAR_SIG, SLASH_SIG
};

enum State {                                // enumeration for CParser states
   CODE, SLASH, COMMENT, STAR
};

class CParser1 {
private:
   State myState;                           // the scalar state-variable
   long myCommentCtr;                       // comment character counter
   /* ... */                                // other CParser1 attributes
public:
   void init() { myCommentCtr = 0; tran(CODE); }       // default transiton
   void dispatch(unsigned const sig);
   void tran(State target) { myState = target; }
   long getCommentCtr() const { return myCommentCtr; }
};
```

state machine interface

The Nested `switch` Statement: Example

first switching the states

then switch signals inside each state

finally the business logic

```
void CParser1::dispatch(unsigned const sig) {
   switch (myState) {
   case CODE:
      switch (sig) {
      case SLASH_SIG:
         tran(SLASH); // transition to SLASH
         break;
      }
      break;
   case SLASH:
      switch (sig) {
      case STAR_SIG:
         myCommentCtr += 2;
         tran(COMMENT);
         break;
      case CHAR_SIG:
      case SLASH_SIG:
         tran(CODE);
         break;
      }
      break;
   case COMMENT:
      switch (sig) {
      case STAR_SIG:
         tran(STAR);
         break;
      case CHAR_SIG:
      case SLASH_SIG:
         ++myCommentCtr;
         break;
      }
      break;
```

The Nested `switch` Statement: Example

one more time, for state STAR

```
   case STAR:
      switch (sig) {
      case STAR_SIG:
         ++myCommentCtr;                    // count STAR as comment
         break;
      case SLASH_SIG:
         myCommentCtr += 2;           // count STAR-SLASH as comment
         tran(CODE);                           // transition to CODE
         break;
      case CHAR_SIG:
         myCommentCtr += 2;               // count STAR-? as comment
         tran(COMMENT);                        // go back to COMMENT
         break;
      }
      break;
   }
}
```


The Nested `switch` Statement

- Nest switch statement with
  - a scalar state variable in the first level for states,
  - an event signal in the second level, and
  - transition logic (actions) in the innermost level
- Advantages
  - Simple and straightforward – just enumeration of stats and triggers
  - Small memory footprint – only a state variable necessary
- Disadvantages
  - Does not promote code reuse
    - all elements of an EFSM must be coded specifically for problem at hand.
  - Manual code is prone to errors
    - when logic becomes complex
  - Difficult to maintain in view of design change

State Table

- A table of arrays of transitions for each state
  - function pointers are stored for easy management
  - fast event dispatching
    - store states and signals as integers, then calculate with pointer offsets

```
trans *t = tableAddress + state * numSignals + sig;
```

Signals →

States ↓

	CHAR_SIG	STAR_SIG	SLASH_SIG
code			doNothing(), slash
slash	doNothing(), code	a2(), comment	doNothing(), code
comment	a1(), comment	doNothing(), star	a1(), comment
star	a2(), comment	a1(), star	a2(), code

* a1() and a2() are respective action functions

State Table

- Advantages
  - Direct mapping from a tabular representation of an EFSM
  - Fast event dispatching
  - Code reuse of the "generic event dispatching process"
- Disadvantages
  - Table maybe large and wasteful

Signals →

States ↓

	CHAR_SIG	STAR_SIG	SLASH_SIG
code			doNothing(), slash
slash	doNothing(), code	a2(), comment	doNothing(), code
comment	a1(), comment	doNothing(), star	a1(), comment
star	a2(), comment	a1(), star	a2(), code

* a1() and a2() are respective action functions

State Table: Example

a generic state table class

struct for transitions

struct for transitions

fast dispatching

```
class StateTable {
public:
   typedef void (StateTable::*
   struct Tran {
      Action action;
      unsigned nextState;
   };
   StateTable(Tran const *table, unsigned nStates, unsigned nSignals)
      : myTable(table), myNsignals(nSignals), myNstates(nS
   virtual ~StateTable() {}                           // vi
   void dispatch(unsigned const sig) {
      register Tran const *t = myTable + myState*myNsignals + sig;
      (this->*(t->action))();
      myState = t->nextState;
   }
   void doNothing() {}
protected:
   unsigned myState;
};
```

```
private:
   Tran const *myTable;
   unsigned myNsignals;
   unsigned myNstates;
```


State Table: Example

table size

constructing customized state table

```
// specific Comment Parser state machine ...
enum Event { CHAR_SIG, STAR_SIG, SLASH_SIG, MAX_SIG };
enum State { CODE, SLASH, COMMENT, STAR, MAX_STATE };

class CParser2 : public StateTable {                  // CPa
public:
   CParser2() : StateTable(&myTable[0][0], MAX_STATE, MAX_SIG) {}
   void init() { myCommentCtr = 0; myState = CODE; }           // initial tran
   long getCommentCtr() const { return myCommentCtr; }
private:
   void a1() { myCommentCtr += 1; }                            // action method
   void a2() { myCommentCtr += 2; }                            // action method
private:
   static StateTable::Tran const myTable[MAX
   long myCommentCtr;                                   character counter
};
```

business logic implemented as private functions of the table

static field holding the state table

State Table: Example

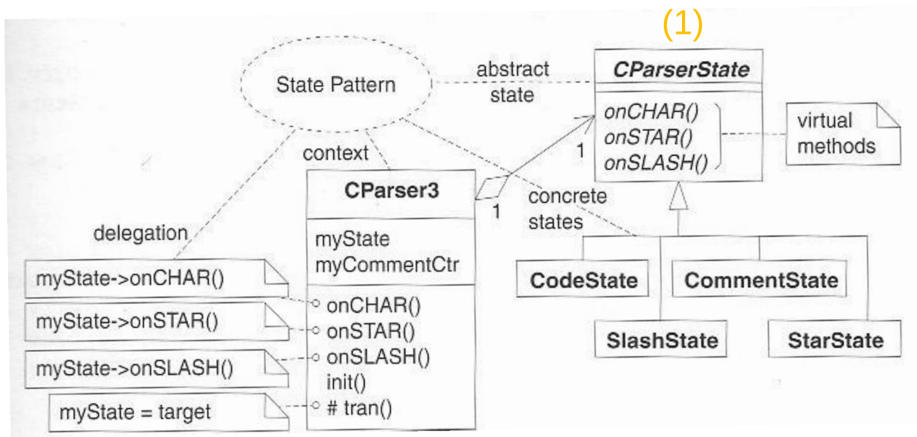
filling out the table statically

```
#include "cparser2.h"

StateTable::Tran const CParser2::myTable[MAX_STATE][MAX_SIG] = {
   {{&StateTable::doNothing, CODE },
    {&StateTable::doNothing, CODE },
    {&StateTable::doNothing, SLASH}},

   {{&StateTable::doNothing, CODE },
    {static_cast<StateTable::Action>(&CParser2::a2), COMMENT },
    {&StateTable::doNothing, CODE }},

   {{static_cast<StateTable::Action>(&CParser2::a1), COMMENT },
    {&StateTable::doNothing,STAR  },
    {static_cast<StateTable::Action>(&CParser2::a1), COMMENT }},

   {{static_cast<StateTable::Action>(&CParser2::a2), COMMENT },
    {static_cast<StateTable::Action>(&CParser2::a1), STAR },
    {static_cast<StateTable::Action>(&CParser2::a2), CODE }}
};
```


State Design Pattern [Gamma+ 95, Douglass 99]

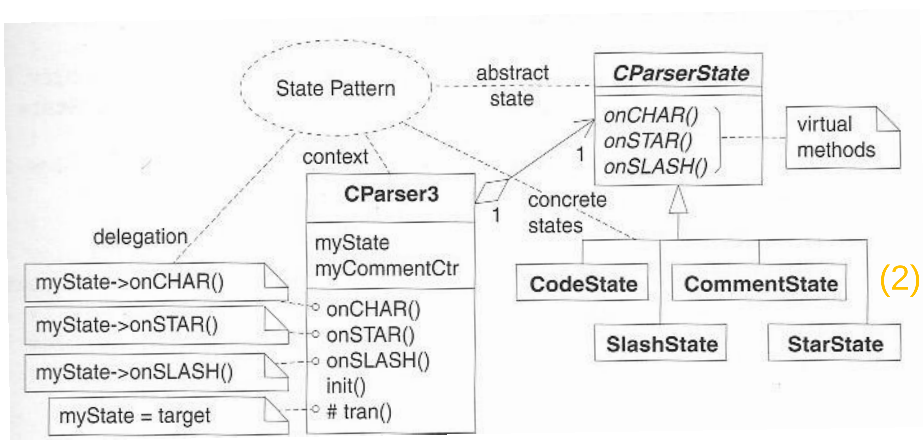
- An abstract state class
  - defines a common interface for handling events
  - each event corresponds to a virtual method
- Concrete states are subclasses of an abstract state class
- A context class delegates all events for processing to the current state object (myState variable)
- State transitions are explicit and are accomplished by reassigning the (myState variable)
- Adding new events corresponds adding it to the abstract state class
- Adding new states is to subclass the abstract state class

State Design Pattern [Gamma+ 95, Douglass 99]



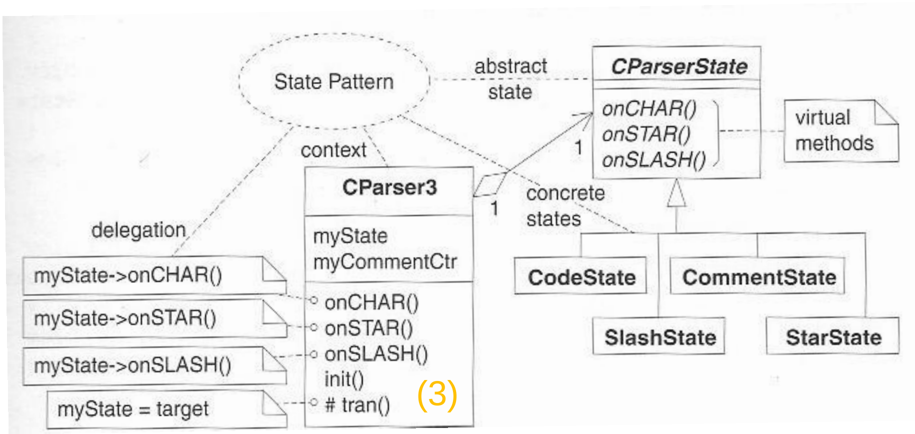
(1) A class for abstract states, contains all possible signals as virtual methods.

State Design Pattern [Gamma+ 95, Douglass 99]



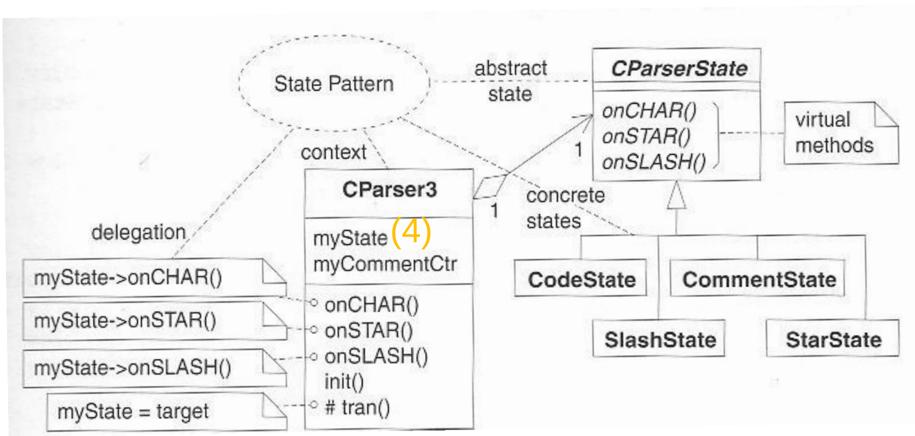
(2) A state in an EFSM is an object of a subclass of the CParserState. (CodeState, SlashState, etc.)

State Design Pattern [Gamma+ 95, Douglass 99]



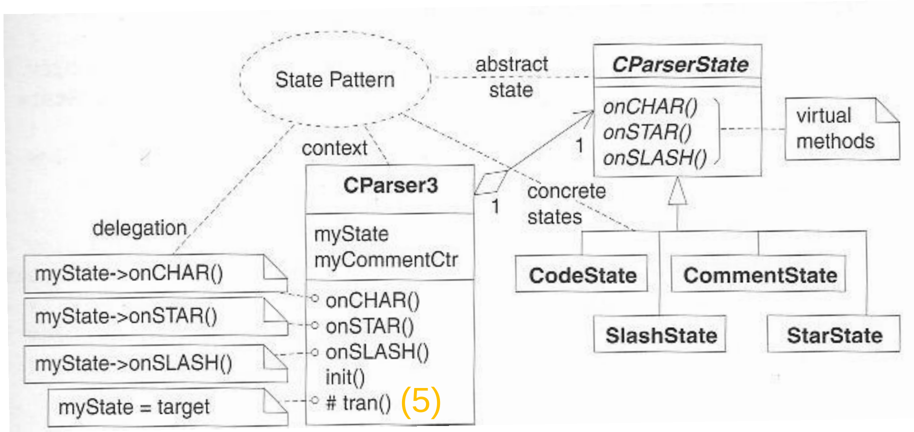
(3) The parser has exactly one object for each concrete state class (e.g., myCodeState, mySlashState, etc.)…

State Design Pattern [Gamma+ 95, Douglass 99]



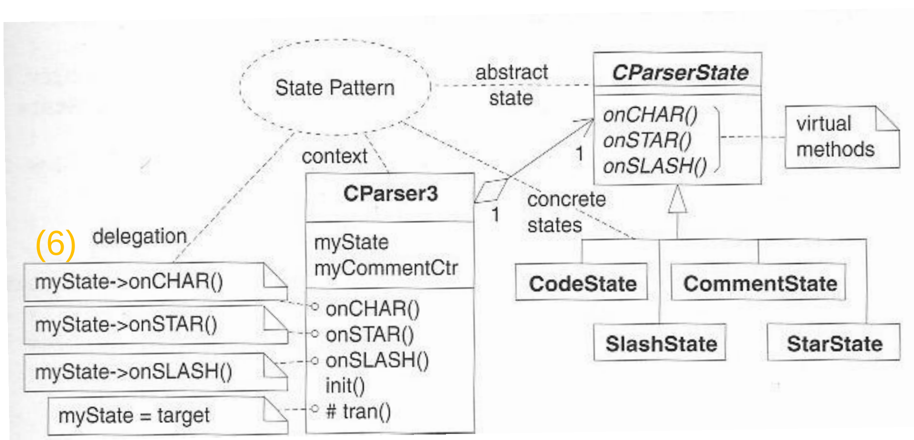
(4) … as well as a myState variable, which can points to any of them.

State Design Pattern [Gamma+ 95, Douglass 99]



(5) State transition is just reassigning myState variable.

State Design Pattern [Gamma+ 95, Douglass 99]



(6) Event dispatching relies on C++ virtual function dispatching rules. (If concrete class has the method, use it; otherwise use the virtual methods.) – Fast dispatching.

State Design Pattern: Example

The same story once more, in code.

```
class CParserState {                                  // abstract State
public:
    virtual void onCHAR(CParser3 *context, char ch)  {}
    virtual void onSTAR(CParser3 *context) {}
    virtual void onSLASH(CParser3 *context) {}
};
class CodeState : public CParserState {
public:
    virtual void onSLASH(CParser3 *conte
};
class SlashState : public CParserState
public:
    virtual void onCHAR(CParser3 *context, char ch);
    virtual void onSTAR(CParser3 *context);
};
class CommentState : public CParserState
    ....
};
class StarState : public CParserState {
    ....
};
```

(1) A class for abstract states, contains all possible signals as virtual methods.

(2) A state in an EFSM is an object of a subclass of the CParserState. (CodeState, SlashState, etc.)

State Design Pattern: Example

The same story once more, in code.

```
class CParser3 {                                        // Context class
   friend class CodeState;
   friend class SlashState;
   friend class CommentState;
   friend class StarState;
   static CodeState     myCodeState;
   static SlashState    mySlashState;
   static CommentState  myCommentState;
   static StarState     myStarState;

   CParserState *myState;
   long myCommentCtr;
public:
   void init() { myCommentCtr = 0; tran(&myCodeS
   void tran(CParserState *target) { myState = t
   long getCommentCtr() const { return myComment
   void onCHAR(char ch) { myState->onCHAR(this,
   void onSTAR() { myState->onSTAR(this);  }
   void onSLASH() { myState->onSLASH(this); }
};
```

(3) The parser has exactly one object for each concrete state class (e.g., myCodeState, mySlashState

(4) … as well as a myState variable, which can points to

(5) State transition is just

(6) Event dispatching relies on C++ virtual function dispatching rules. (If concrete class has the method, use it; otherwise use the virtual methods.) – Fast dispatching.

State Design Pattern: Example

The same story once more, in code.

```
#include "cparser3.h"

CodeState    CParser3::myCodeState;
SlashState   CParser3::mySlashState;
CommentState CParser3::myCommentState;
StarState    CParser3::myStarState;

void CodeState::onSLASH(CParser3 *context) {
   context->tran(&CParser3::mySlashState);
}
void SlashState::onCHAR(CParser3 *context, char ch) {
   context->tran(&CParser3::myCodeState);
}
....
void StarState::onSTAR(CParser3 *context) {
   context->myCommentCtr++;
}
void StarState::onSLASH(CParser3 *context) {
   context->myCommentCtr += 2;
   context->tran(&CParser3::myCodeState);
}
```

(7) Implementation of individual actions are writing transition functions on need. (Default behavior is `doNothing()` as in the virtual function.)

State Design Pattern [Gamma+ 95, Douglass 99]

- Advantages
  - It localizes state specific behavior in separate (sub)classes.
  - Efficient state transition – reassigning a pointer.
  - Fast event dispatching – using C++ mechanism for function look up.
  - Parameterized events made easy – passing function parameters.
  - No need for enumeration of states or events beforehand.
- Disadvantages
  - Adding states requires creating subclasses.
  - Adding new events requires adding handlers to state interface.
  - In some situations where C++ or Java is not supported (e.g., some embedded systems), mockups of OO design maybe an overkill.

Multiple-threaded Implementation

- Approach I
  - Each EFSM is implemented inside one thread.
  - Threads run simultaneously, scheduled in round-robin.
  - EFSMs share variables in the process.
- Advantage
  - Straightforward transformation from model.
  - EFSM communication easily implemented with thread messages.
- Disadvantage
  - In some situations, no ready thread support in specific platform.
  - Related analysis (progressiveness if semaphores are used, timing properties, etc) may be difficult.

Multiple-threaded Implementation

- Approach II
  - Event detectors are implemented in threads.
  - Transition actions are implemented in functions.
  - Location information is stored with a variable.
  - When event detector threads detects, calls corresponding functions and switching locations.
- Advantage
  - Easy adaption to model changes.
- Disadvantage
  - In some situations, no ready thread support in specific platform.
  - Code may be unstructured/unreadable.

Multiple-threaded Implementation: Example (Approach II)

```
void *trans3(void *ptr) {
    int t;
    while(1) {
        sem_wait(&Sense);
        if(current==WAIT_VRP && TRUE) {
            t=getTimer(&v_x);
            printf("Sense:%d\n", t);
            clearTimer(&v_x);
            current=ST_IDLE;
            sem_post(&ST_IDLE);
        }
    }
}
int main(int argc, char *argv[]) {
    pthread_t thread1, thread2,thread3;
    // ... initilization code ...
    pthread_create( &thread1, NULL, trans1, NULL);
    pthread_create( &thread2, NULL, trans2, NULL);
    pthread_create( &thread3, NULL, trans3, NULL);
    pthread_join( thread1, NULL);
    return 0;
}
```

- A transition logic is written inside a function
  - Wait for semaphore for triggering signal.
  - If succeeded, check state (WAIT_VRP) and guard (TRUE).
  - Execute updates.
    - Print out timer value (for debugging)
    - Reset timer value
    - Change state
- All threads initialized and run in main

Optimal EFSM Implementation

- Does there exist one?
- A trade off based on
  - platform (available libraries? languages to use?)
  - purpose of coding
    - just for implementation or for analysis?
    - what type of analysis? etc.
  - efficiency requirement
  - possibility of model changes

THE EFSM TOOLSET

The EFSM Toolset

- Introduction
- The EFSM Language
- Checking for Non-determinism and Totality
- Translations to Other Languages
- Test Generation from EFSMs
- Script Generation
- Code Generation
- Simulation

Introduction to EFSM Toolset

- Targets designers and engineers without specialized training in formal methods
- Uses easily human-readable languages in description
- Features
  - based on communicating EFSMs
  - using communication channels as well as shared variables
  - with input, output, and local variables

The EFSM Language

Channels and channel actions

- Example

```
SYSTEM LampSystem:
Channel: press, sync, B;
EFSM Lamp {
  States: off, low, bright;
  InitialState: off;
  LocalVars:
    bint[0..5] y =0,
    boolean x = false;
  Transition: From off to low when true==true do press ? x , y =0;
  Transition: From low to off when y>=5 do press ? x;
  Transition: From low to bright when y<5 do press ? x;
  Transition: From bright to off when true==true do press ? x;
}
EFSM User {
  States: idle;
  InitialState: idle;
  LocalVars:
    boolean x = false;
  Transition: From idle to idle when true==true do press ! x;
}
```

The EFSM Language: Channels

- Specification: `Channel: <name>, <type>, <r/w/b>;`
  - `<type>` maybe "sync" or "async"
    - "sync": one-to-one, blocking synchronization of two EFSMs
    - "async": one-writer-to-many-readers, non-blocking for writer, non-consuming from the readers, asynchronous
  - `<r/w/b>` – reader channel, writer channel, or both
- Communication action: `<channel name> <action> <arg>`
  - `<action>` can be !, ?, !!, ??
    - ! means writing, ? means reading
    - single mark means synchronous; double marks means asynchronous
  - `<arg>` is a typeless value

Non-determinism and Totality

- Definitions
  - An EFSM is non-determinism if there exists some state with two or more outgoing transitions which maybe enabled at the same time.
  - An EFSM is total (or complete) if at any state S, there is at least one outgoing edge enabled.
- Checking Non-determinism and Totality
  - The EFSM Toolset utilizes the satisfiability checker zChaff
    - systematically transform logic formula over guards to DIMACS (zChaff accepted format)
    - feed into zChaff and interpret the result

Non-determinism and Totality

- Example: A system with a bounded integer x in [1..5] and
  (1) `From start to discard if x<=4;`
  (2) `From start to use if x>3;`
  (3) `From start to redo if x==5;`
- To check for non-determinism for the "start" state, we need to check
  (a) $(x<=4) \wedge (x>3)$
  (b) $(x<=4) \wedge (x==5)$
  (c) $(x>3) \wedge (x==5)$

  which are transformed to (xk means x==k is true)
  (a) $(x1 \vee x2 \vee x3 \vee x4) \wedge (x4 \vee x5)$
  (b) $(x1 \vee x2 \vee x3 \vee x4) \wedge (x5)$
  (c) $(x4 \vee x5) \wedge (x5)$
- (a), (c) are satisfied, which means there are non-determinisms between
  - the first and the second transitions, and
  - the second and the third transitions

Translations to Other Languages

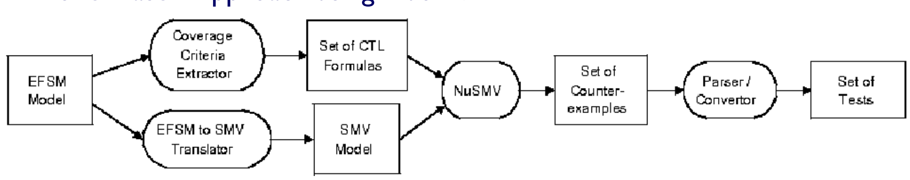
- Table – CSV format most Spreadsheet programs can open
- Text – Human readable text file
- Dotty – Input format of the "dot" program for drawing
- Promela – The Spin model checker input language
  - systems with only synchronous channels are supported now
  - setting channel buffer size to 0 in Promela
- Uppaal – The Uppaal model checker XML format
  - systems with only synchronous channels are supported now
  - value passing must take place through shared variables
- SMV format – the NuSMV model checker
  - currently supports systems without communication channels
  - shared variables are supported
  - can utilize the verification results from NuSMV for test case generation

Test Generation from EFSMs

- Test Generation
  - create a set of traces through the EFSM
  - for testing an implementation of the system
  - coverage criteria
    - state coverage – go through each state at least once
    - transition coverage – ensures that every transition is take once
    - definition-use coverage – makes a trace from every definition of a variable to each of its subsequent uses
- Output of Test Generation
  - a sequence of assignments of variable values
  - necessary to lead the EFSM to take particular paths

Test Generation from EFSMs

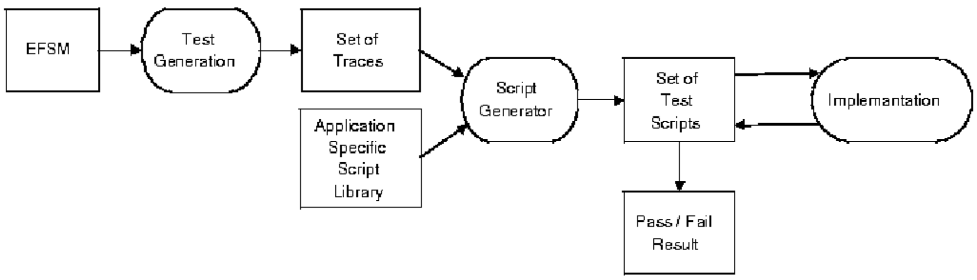
- Model Based Approach using NuSMV



- Direct Test Generation Algorithm
  - DFS algorithm to discover set of paths covering all states
  - It keeps track of constraints and assignments along the path.
  - A transition can be added to the path if compatible with other constraints.
  - The algorithm starts with initial state and a set of constraints corresponding to the initial assignments of variables.
  - If a transition can be added (compatible), add it and make a recursive call.
  - Can keep a snapshot of visited states to guarantee termination.
  - Need to use a satisfiability checker.

Script Generation

- Script Generation
  - takes a set of traces (from test generation)
  - makes a set of scripts
  - which can be run on an implementation
- Approach



Code Generation

- Allows mapping state machine variables to arbitrary input/output functions
  - input function is called each time the variable is read
    - condition like x==2 will be translated to readX()==2
  - output function is called each time the variable is written
    - assignment like x:=y-2 will become writeX(y-2)
- C
  - limited to single EFSM model without communication channels
  - repeatedly executing a series of if-else statements like
    - if (state==X and condition==c) { doSomething(); … }
- Java
  - supports multiple communicating machines, utilizing JCSP library
  - each EFSM is run inside a thread, and they are all run in parallel
  - scheduled round-robin

Example: Generated Code Snippet in C

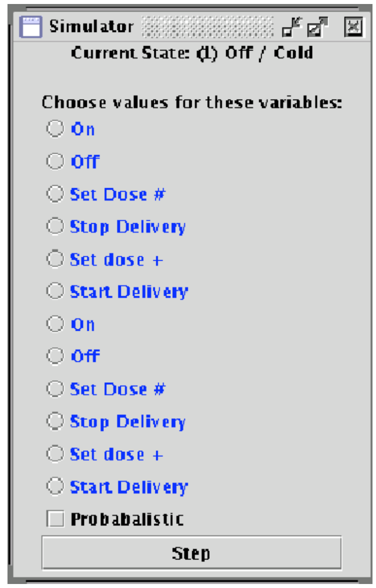
```
while (transition_taken) {
    transition_taken=0;
    if ((currentState==1)&&(current < getNumBots())) {
        PRE_TRANSITION_HOOK;
        current=current+1;
        currentState=2;
        transition_taken=1;
        //printf("rolling:From:incrementTo:check
        //Guard:current <=numBotsAction:current=current+1\n");
        POST_TRANSITION_HOOK;
    }
    ....
    if ((currentState==2)&&(getSwitch()==1)) {
        PRE_TRANSITION_HOOK;
        setAllAngles(1);
        currentState=1;
        transition_taken=1;
        //printf("rolling:From:checkTo:increment
        //Guard:switchValue==trueAction:setAngles=true \n");
        POST_TRANSITION_HOOK;
    }
}
```


Example: Generated Code Snippet in Java

```
import jcsp.lang.*;
public class gaitThread implements CSProcess {
    private final String name="gaitThread"; ....
    public gaitThread() {}
    public void run() {
        final Skip skip= new Skip(); ....
        while ( true ) {
            switch (alt.priSelect()) {
            case 0:
                if ((currentState.equals("footArming")) & (TwoSecondDelay == true )) {
                    currentState="spineArming";
                    System.out.println("gait: From: footArming
                                        To:spineArming
                                        Guard:TwoSecondDelay== true
                                        Action:spine=-20");
                }
                if ((currentState.equals("spineArming")) & (TwoSecondDelay == true)) {
                     ....
                } ....
            ....
            }
        }
    }
}
```

Simulation with the EFSM Toolset

- Simulator generates Java code which walks through the EFSM model.
- At each step, it asks the user for input variable values, if any.
- Example screen on the right.
  - Either choose variables to set value and "Step", or
  - Auto with "Probabilistic".
- Simulator reports error message if no transitions available.



References

- Miro Samek, *Practical Statecharts in C/C++*, CMPBooks, 2002
  - Source code available online
  - http://www.state-machine.com/psicc/
- David Arney, *EFSM Toolbox Manual*, University of Pennsylvania, 2009

Thank you!