**Automata-based programming**

From Wikipedia, the free encyclopedia

Jump to: [navigation](http://en.wikipedia.org/wiki/Automata-based_programming#mw-head), [search](http://en.wikipedia.org/wiki/Automata-based_programming#p-search)

For other uses, see [Automata-based programming (Shalyto's approach)](http://en.wikipedia.org/wiki/Automata-based_programming_%28Shalyto%27s_approach%29).

|  |
| --- |
| [**Programming paradigms**](http://en.wikipedia.org/wiki/Programming_paradigm) |
| * [Action](http://en.wikipedia.org/wiki/Action_language) * [Agent-oriented](http://en.wikipedia.org/wiki/Agent-oriented_programming) * [Aspect-oriented](http://en.wikipedia.org/wiki/Aspect-oriented_programming) * **Automata-based** * [Component-based](http://en.wikipedia.org/wiki/Component-based_software_engineering)   + [Flow-based](http://en.wikipedia.org/wiki/Flow-based_programming)   + [Pipelined](http://en.wikipedia.org/wiki/Pipeline_programming) * [Concatenative](http://en.wikipedia.org/wiki/Concatenative_programming_language) * [Concurrent computing](http://en.wikipedia.org/wiki/Concurrent_computing)   + [Relativistic programming](http://en.wikipedia.org/wiki/Relativistic_programming) * [Data-driven](http://en.wikipedia.org/wiki/Data-driven_programming) * [Declarative](http://en.wikipedia.org/wiki/Declarative_programming) (contrast: [Imperative](http://en.wikipedia.org/wiki/Imperative_programming))   + [Constraint](http://en.wikipedia.org/wiki/Constraint_programming)   + [Dataflow](http://en.wikipedia.org/wiki/Dataflow_programming)     - Cell-oriented ([spreadsheets](http://en.wikipedia.org/wiki/Spreadsheet))     - [Reactive](http://en.wikipedia.org/wiki/Reactive_programming)     - [Intensional](http://en.wikipedia.org/wiki/Category:Intensional_programming_languages)   + [Functional](http://en.wikipedia.org/wiki/Functional_programming)   + [Logic](http://en.wikipedia.org/wiki/Logic_programming)     - [Abductive logic](http://en.wikipedia.org/wiki/Abductive_logic_programming)     - [Answer set](http://en.wikipedia.org/wiki/Answer_set_programming)     - [Constraint logic](http://en.wikipedia.org/wiki/Constraint_logic_programming)     - [Functional logic](http://en.wikipedia.org/wiki/Functional_logic_programming)     - [Inductive logic](http://en.wikipedia.org/wiki/Inductive_logic_programming) * [End-user programming](http://en.wikipedia.org/wiki/End-user_development) * [Event-driven](http://en.wikipedia.org/wiki/Event-driven_programming)   + [Service-oriented](http://en.wikipedia.org/wiki/Service-oriented_architecture)   + [Time-driven](http://en.wikipedia.org/wiki/Time-driven_programming) * [Expression-oriented](http://en.wikipedia.org/wiki/Expression-oriented_programming_language) * [Feature-oriented](http://en.wikipedia.org/wiki/Feature_Oriented_Programming) * [Function-level](http://en.wikipedia.org/wiki/Function-level_programming) (contrast: [Value-level](http://en.wikipedia.org/wiki/Value-level_programming)) * [Generic](http://en.wikipedia.org/wiki/Generic_programming) * [Imperative](http://en.wikipedia.org/wiki/Imperative_programming) (contrast: [Declarative](http://en.wikipedia.org/wiki/Declarative_programming))   + [Procedural](http://en.wikipedia.org/wiki/Procedural_programming) * [Language-oriented](http://en.wikipedia.org/wiki/Language-oriented_programming)   + [Discipline-specific](http://en.wikipedia.org/wiki/Service-oriented_modeling#Discipline-specific_modeling)   + [Domain-specific](http://en.wikipedia.org/wiki/Domain-specific_language)   + [Grammar-oriented](http://en.wikipedia.org/wiki/Grammar-oriented_programming)     - [Dialecting](http://en.wikipedia.org/wiki/Dialecting)   + [Intentional](http://en.wikipedia.org/wiki/Intentional_programming) * [Metaprogramming](http://en.wikipedia.org/wiki/Metaprogramming)   + [Automatic](http://en.wikipedia.org/wiki/Automatic_programming)   + [Reflective](http://en.wikipedia.org/wiki/Reflection_%28computer_programming%29)     - [Attribute-oriented](http://en.wikipedia.org/wiki/Attribute-oriented_programming)   + [Template](http://en.wikipedia.org/wiki/Template_metaprogramming)     - [Policy-based](http://en.wikipedia.org/wiki/Policy-based_design) * [Non-structured](http://en.wikipedia.org/wiki/Non-structured_programming) (contrast: [Structured](http://en.wikipedia.org/wiki/Structured_programming))   + [Array](http://en.wikipedia.org/wiki/Array_programming) * [Nondeterministic](http://en.wikipedia.org/wiki/Nondeterministic_programming) * [Parallel computing](http://en.wikipedia.org/wiki/Parallel_computing)   + [Process-oriented](http://en.wikipedia.org/wiki/Process-oriented_programming) * [Programming in the large and small](http://en.wikipedia.org/wiki/Programming_in_the_large_and_programming_in_the_small) * [Semantic](http://en.wikipedia.org/wiki/Semantic-oriented_programming) * [Structured](http://en.wikipedia.org/wiki/Structured_programming) (contrast: [Non-structured](http://en.wikipedia.org/wiki/Non-structured_programming))   + [Modular](http://en.wikipedia.org/wiki/Modular_programming) (contrast: Monolithic)   + [Object-oriented](http://en.wikipedia.org/wiki/Object-oriented_programming)     - By [separation of concerns](http://en.wikipedia.org/wiki/Separation_of_concerns):       * [Aspect-oriented](http://en.wikipedia.org/wiki/Aspect-oriented_programming)       * [Role-oriented](http://en.wikipedia.org/wiki/Role-oriented_programming)       * [Subject-oriented](http://en.wikipedia.org/wiki/Subject-oriented_programming)     - [Class-based](http://en.wikipedia.org/wiki/Class-based_programming)     - [Prototype-based](http://en.wikipedia.org/wiki/Prototype-based_programming)   + [Recursive](http://en.wikipedia.org/wiki/Recursion_%28computer_science%29) * [Value-level](http://en.wikipedia.org/wiki/Value-level_programming) (contrast: [Function-level](http://en.wikipedia.org/wiki/Function-level_programming)) |
| * [v](http://en.wikipedia.org/wiki/Template:Programming_paradigms) * [t](http://en.wikipedia.org/wiki/Template_talk:Programming_paradigms) * [e](http://en.wikipedia.org/w/index.php?title=Template:Programming_paradigms&action=edit) |

**Automata-based programming** is a [programming paradigm](http://en.wikipedia.org/wiki/Programming_paradigm) in which the program or its part is thought of as a model of a [finite state machine](http://en.wikipedia.org/wiki/Finite_state_machine) (FSM) or any other (often more complicated) formal automaton (see [automata theory](http://en.wikipedia.org/wiki/Automata_theory)). Sometimes a potentially-infinite set of possible states is introduced, and such a set can have a complicated structure, not just an enumeration.

**FSM-based programming** is generally the same, but, formally speaking, doesn't cover all possible variants as FSM stands for [finite state machine](http://en.wikipedia.org/wiki/Finite_state_machine) and **automata-based programming** doesn't necessarily employ FSMs in the strict sense.

The following properties are key indicators for automata-based programming:

1. The time period of the program's execution is clearly separated down to the *steps of the automaton*. Each of the *steps* is effectively an execution of a code section (same for all the steps), which has a single entry point. Such a section can be a function or other routine, or just a cycle body. The step section might be divided down to subsection to be executed depending on different states, although this is not necessary.
2. Any communication between the steps is only possible via the explicitly noted set of variables named *the state*. Between any two steps, the program (or its part created using the automata-based technique) can **not** have implicit components of its state, such as local (stack) variables' values, return addresses, the current instruction pointer etc. That is, the state of the whole program, taken at any two moments of entering the step of the automaton, can only differ in the values of the variables being considered as the state of the automaton.

The whole execution of the automata-based code is a (possibly explicit) cycle of the automaton's steps.

Another reason to use the notion of **automata-based programming** is that the programmer's style of thinking about the program in this technique is very similar to the style of thinking used to solve math-related tasks using [Turing machine](http://en.wikipedia.org/wiki/Turing_machine), [Markov algorithm](http://en.wikipedia.org/wiki/Markov_algorithm) etc.

|  |
| --- |
| **Contents**   * [1 Example](http://en.wikipedia.org/wiki/Automata-based_programming#Example)   + [1.1 Traditional (imperative) program in C](http://en.wikipedia.org/wiki/Automata-based_programming#Traditional_.28imperative.29_program_in_C)   + [1.2 Automata-based style program](http://en.wikipedia.org/wiki/Automata-based_programming#Automata-based_style_program)   + [1.3 A separate function for the automation step](http://en.wikipedia.org/wiki/Automata-based_programming#A_separate_function_for_the_automation_step)   + [1.4 Explicit state transition table](http://en.wikipedia.org/wiki/Automata-based_programming#Explicit_state_transition_table)   + [1.5 Automation and Automata](http://en.wikipedia.org/wiki/Automata-based_programming#Automation_and_Automata)     - [1.5.1 Example Program](http://en.wikipedia.org/wiki/Automata-based_programming#Example_Program)     - [1.5.2 Automation & Events](http://en.wikipedia.org/wiki/Automata-based_programming#Automation_.26_Events)   + [1.6 Using object-oriented capabilities](http://en.wikipedia.org/wiki/Automata-based_programming#Using_object-oriented_capabilities) * [2 Applications](http://en.wikipedia.org/wiki/Automata-based_programming#Applications) * [3 History](http://en.wikipedia.org/wiki/Automata-based_programming#History) * [4 Compared against imperative and procedural programming](http://en.wikipedia.org/wiki/Automata-based_programming#Compared_against_imperative_and_procedural_programming) * [5 Object-oriented programming relationship](http://en.wikipedia.org/wiki/Automata-based_programming#Object-oriented_programming_relationship) * [6 See also](http://en.wikipedia.org/wiki/Automata-based_programming#See_also) * [7 References](http://en.wikipedia.org/wiki/Automata-based_programming#References) * [8 See also](http://en.wikipedia.org/wiki/Automata-based_programming#See_also_2) * [9 External links](http://en.wikipedia.org/wiki/Automata-based_programming#External_links) |

**Example**

Consider a program in [C](http://en.wikipedia.org/wiki/C_%28programming_language%29) that reads a text from *standard input stream*, line by line, and prints the first word of each line. It is clear we need first to read and skip the leading spaces, if any, then read characters of the first word and print them until the word ends, and then read and skip all the remaining characters until the end-of-line character is encountered. Upon reaching the end of line character (regardless of the stage), we restart the algorithm from the beginning, and upon encountering the *end of file* condition (regardless of the stage), we terminate the program.

**Traditional (imperative) program in C**

The program which solves the example task in traditional ([imperative](http://en.wikipedia.org/wiki/Imperative_programming)) style can look something like this:

#include <stdio.h>

int main(void)

{

int c;

do {

c = getchar();

while(c == ' ')

c = getchar();

while(c != EOF && c != ' ' && c != '\n') {

putchar(c);

c = getchar();

}

putchar('\n');

while(c != EOF && c != '\n')

c = getchar();

} while(c != EOF);

return 0;

}

**Automata-based style program**

The same task can be solved by thinking in terms of finite state machines. Note that line parsing has three stages: skipping the leading spaces, printing the word and skipping the trailing characters. Let's call them **states** before, inside and after. The program may now look like this:

#include <stdio.h>

int main(void)

{

enum states {

before, inside, after

} state;

int c;

state = before;

while((c = getchar()) != EOF) {

switch(state) {

case before:

if(c == '\n') {

putchar('\n');

} else

if(c != ' ') {

putchar(c);

state = inside;

}

break;

case inside:

switch(c) {

case ' ': state = after; break;

case '\n':

putchar('\n');

state = before;

break;

default: putchar(c);

}

break;

case after:

if(c == '\n') {

putchar('\n');

state = before;

}

}

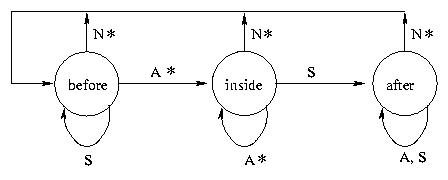
}

return 0;

}

Although the code now looks longer, it has at least one significant advantage: there's only one *reading* (that is, call to the getchar() function) instruction in the program. Besides that, there's only one loop instead of the four the previous versions had.

In this program, the body of the while loop is the **automaton step**, and the loop itself is the *cycle of the automaton's work*.

[](http://en.wikipedia.org/wiki/File:Automata_that_prints_the_first_word_of_each_line.png)

The program implements (models) the work of a *finite state machine* shown on the picture. The **N** denotes the end of line character, the **S** denotes spaces, and the **A** stands for all the other characters. The automaton follows exactly one *arrow* on each step depending on the current state and the encountered character. Some state switches are accompanied with printing the character; such arrows are marked with asterisks.

It is not absolutely necessary to divide the code down to separate handlers for each unique state. Furthermore, in some cases the very notion of the *state* can be composed of several variables' values, so that it could be impossible to handle each possible state explicitly. In the discussed program it is possible to reduce the code length by noticing that the actions taken in response to the end of line character are the same for all the possible states. The following program is equal to the previous one but is a bit shorter:

#include <stdio.h>

int main(void)

{

enum states {

before, inside, after

} state;

int c;

state = before;

while((c = getchar()) != EOF) {

if(c == '\n') {

putchar('\n');

state = before;

} else

switch(state) {

case before:

if(c != ' ') {

putchar(c);

state = inside;

}

break;

case inside:

if(c == ' ') {

state = after;

} else {

putchar(c);

}

break;

case after:

break;

}

}

return 0;

}

**A separate function for the automation step**

The most important property of the previous program is that the automaton step code section is clearly localized. It is possible to demonstrate this property even better if we provide a separate function for it:

#include <stdio.h>

enum states { before, inside, after };

void step(enum states \*state, int c)

{

if(c == '\n') {

putchar('\n');

\*state = before;

} else

switch(\*state) {

case before:

if(c != ' ') {

putchar(c);

\*state = inside;

}

break;

case inside:

if(c == ' ') {

\*state = after;

} else {

putchar(c);

}

break;

case after:

break;

}

}

int main(void)

{

int c;

enum states state = before;

while((c = getchar()) != EOF) {

step(&state, c);

}

return 0;

}

This example clearly demonstrates the basic properties of automata-based code:

1. time periods of automaton step executions may not overlap
2. the only information passed from the previous step to the next is the explicitly specified *automaton state*

**Explicit state transition table**

A finite automaton can be defined by an explicit [state transition table](http://en.wikipedia.org/wiki/State_transition_table). Generally speaking, an automata-based program code can naturally reflect this approach. In the program below there's an array named the\_table, which defines the table. The rows of the table stand for three *states*, while columns reflect the input characters (first for spaces, second for the end of line character, and the last is for all the other characters).

For every possible combination, the table contains the new state number and the flag, which determines whether the automaton must print the symbol. In a real life task, this could be more complicated; e.g., the table could contain pointers to functions to be called on every possible combination of conditions.

#include <stdio.h>

enum states { before = 0, inside = 1, after = 2 };

struct branch {

unsigned char new\_state:2;

unsigned char should\_putchar:1;

};

struct branch the\_table[3][3] = {

/\* ' ' '\n' others \*/

/\* before \*/ { {before,0}, {before,1}, {inside,1} },

/\* inside \*/ { {after, 0}, {before,1}, {inside,1} },

/\* after \*/ { {after, 0}, {before,1}, {after, 0} }

};

void step(enum states \*state, int c)

{

int idx2 = (c == ' ') ? 0 : (c == '\n') ? 1 : 2;

struct branch \*b = & the\_table[\*state][idx2];

\*state = (enum states)(b->new\_state);

if(b->should\_putchar) putchar(c);

}

int main(void)

{

int c;

enum states state = before;

while((c = getchar()) != EOF)

step(&state, c);

return 0;

}

**Automation and Automata**

Automata-based programming indeed closely matches the programming needs found in the field of [automation](http://en.wikipedia.org/wiki/Automation).

A production cycle is commonly modelled as:

* A sequence of stages stepping according to input data (from captors).
* A set of actions performed depending on the current stage.

Various dedicated programming languages allow expressing such a model in more or less sophisticated ways.

**Example Program**

The example presented above could be expressed according to this view like in the following program. Here [pseudo-code](http://en.wikipedia.org/wiki/Pseudo-code) uses such conventions:

* 'set' and 'reset' respectively activate & inactivate a logic variable (here a stage)
* ':' is assignment, '=' is equality test

SPC : ' '

EOL : '\n'

states : (before, inside, after, end)

setState(c) {

if c=EOF then set end

if before and (c!=SPC and c!=EOL) then set inside

if inside and (c=SPC or c=EOL) then set after

if after and c=EOL then set before

}

doAction(c) {

if inside then write(c)

else if c=EOL then write(c)

}

cycle {

set before

loop {

c : readCharacter

setState(c)

doAction(c)

}

until end

}

The separation of routines expressing cycle progression on one side, and actual action on the other (matching input & output) allows clearer and simpler code.

**Automation & Events**

In the field of automation, stepping from step to step depends on input data coming from the machine itself. This is represented in the program by reading characters from a text. In reality, those data inform about position, speed, temperature, etc. of critical elements of a machine.

Like in [GUI](http://en.wikipedia.org/wiki/GUI) programming, **changes** in the machine state can thus be considered as events causing the passage from a state to another, until the final one is reached. The combination of possible states can generate a wide variety of events, thus defining a more complex production cycle. As a consequence, cycles are usually far to be simple linear sequences. There are commonly parallel branches running together and alternatives selected according to different events, schematically represented below:

s:stage c:condition

s1

|

|-c2

|

s2

|

----------

| |

|-c31 |-c32

| |

s31 s32

| |

|-c41 |-c42

| |

----------

|

s4

**Using object-oriented capabilities**

If the implementation language supports [object-oriented programming](http://en.wikipedia.org/wiki/Object-oriented_programming), a simple refactoring is to [encapsulate](http://en.wikipedia.org/wiki/Encapsulation_%28computer_science%29) the automaton into an object, thus hiding its implementation details. For example, an object-oriented version in [C++](http://en.wikipedia.org/wiki/C%2B%2B) of the same program is below. A more sophisticated refactoring could employ the [State\_pattern](http://en.wikipedia.org/wiki/State_pattern).

#include <stdio.h>

class StateMachine {

enum states { before = 0, inside = 1, after = 2 } state;

struct branch {

enum states new\_state:2;

int should\_putchar:1;

};

static struct branch the\_table[3][3];

public:

StateMachine() : state(before) {}

void FeedChar(int c) {

int idx2 = (c == ' ') ? 0 : (c == '\n') ? 1 : 2;

struct branch \*b = & the\_table[state][idx2];

state = b->new\_state;

if(b->should\_putchar) putchar(c);

}

};

struct StateMachine::branch StateMachine::the\_table[3][3] = {

/\* ' ' '\n' others \*/

/\* before \*/ { {before,0}, {before,1}, {inside,1} },

/\* inside \*/ { {after, 0}, {before,1}, {inside,1} },

/\* after \*/ { {after, 0}, {before,1}, {after, 0} }

};

int main(void)

{

int c;

StateMachine machine;

while((c = getchar()) != EOF)

machine.FeedChar(c);

return 0;

}

Note: To minimize changes not directly related to the subject of the article, the input/output functions from the standard library of [C](http://en.wikipedia.org/wiki/C_%28programming_language%29) are being used. Note the use of the [ternary operator](http://en.wikipedia.org/wiki/Ternary_operator), which could also be implemented as if-else.

**Applications**

Automata-based programming is widely used in [lexical](http://en.wikipedia.org/wiki/Lexical_analysis) and [syntactic analyses](http://en.wikipedia.org/wiki/Syntactic_analysis).[[1]](http://en.wikipedia.org/wiki/Automata-based_programming#cite_note-gram-0)

Besides that, thinking in terms of automata (that is, breaking the execution process down to *automaton steps* and passing information from step to step through the explicit *state*) is necessary for [event-driven programming](http://en.wikipedia.org/wiki/Event-driven_programming) as the only alternative to using parallel processes or threads.

The notions of states and state machines are often used in the field of [formal specification](http://en.wikipedia.org/wiki/Formal_specification). For instance, [UML](http://en.wikipedia.org/wiki/Unified_Modeling_Language)-based software architecture development uses [state diagrams](http://en.wikipedia.org/wiki/State_diagram#UML_state_diagram) to specify the behaviour of the program. Also various [communication protocols](http://en.wikipedia.org/wiki/Communication_protocol) are often specified using the explicit notion of *state* (see, e.g., [RFC 793](http://tools.ietf.org/html/rfc793)[[2]](http://en.wikipedia.org/wiki/Automata-based_programming#cite_note-1)).

Thinking in terms of automata (steps and states) can also be used to describe semantics of some [programming languages](http://en.wikipedia.org/wiki/Programming_language). For example, the execution of a program written in the [Refal](http://en.wikipedia.org/wiki/Refal) language is described as a sequence of *steps* of a so-called abstract Refal machine; the state of the machine is a *view* (an arbitrary Refal expression without variables).

[Continuations](http://en.wikipedia.org/wiki/Continuations) in the [Scheme](http://en.wikipedia.org/wiki/Scheme_%28programming_language%29) language require thinking in terms of steps and states, although Scheme itself is in no way automata-related (it is recursive). To make it possible the [call/cc](http://en.wikipedia.org/wiki/Call/cc) feature to work, implementation needs to be able to catch a whole state of the executing program, which is only possible when there's no implicit part in the state. Such a *caught state* is the very thing called *continuation*, and it can be considered as the *state* of a (relatively complicated) automaton. The step of the automaton is deducing the next continuation from the previous one, and the execution process is the cycle of such steps.

Alexander Ollongren in his book[[3]](http://en.wikipedia.org/wiki/Automata-based_programming#cite_note-2) explains the so-called *Vienna method* of programming languages semantics description which is fully based on formal automata.

The STAT system [[1]](http://www.cs.ucsb.edu/%7Eseclab/projects/stat/index.html) is a good example of using the automata-based approach; this system, besides other features, includes an embedded language called *STATL* which is purely automata-oriented.

**History**

Automata-based techniques were used widely in the domains where there are algorithms based on automata theory, such as formal language analyses.[[1]](http://en.wikipedia.org/wiki/Automata-based_programming#cite_note-gram-0)

One of the early papers on this is by Johnson et al., 1968.[[4]](http://en.wikipedia.org/wiki/Automata-based_programming#cite_note-3)

One of the earliest mentions of automata-based programming as a general technique is found in the paper by [Peter Naur](http://en.wikipedia.org/wiki/Peter_Naur), 1963.[[5]](http://en.wikipedia.org/wiki/Automata-based_programming#cite_note-4) The author calls the technique *Turing machine approach*, however no real [Turing machine](http://en.wikipedia.org/wiki/Turing_machine) is given in the paper; instead, the technique based on states and steps is described.

**Compared against imperative and procedural programming**

The notion of [state](http://en.wikipedia.org/wiki/State_%28computer_science%29) is not exclusive property of automata-based programming.[[6]](http://en.wikipedia.org/wiki/Automata-based_programming#cite_note-5) Generally speaking, *state* (or [program state](http://en.wikipedia.org/wiki/Program_state)) appears during execution of any [computer program](http://en.wikipedia.org/wiki/Computer_program), as a combination of all information that can change during the execution. For instance, a *state* of a traditional [imperative](http://en.wikipedia.org/wiki/Imperative_programming) program consists of

1. values of all variables and the information stored within dynamic memory
2. values stored in registers
3. stack contents (including local variables' values and return addresses)
4. current value of the instruction pointer

These can be divided to the **explicit** part (such as values stored in variables) and the **implicit** part (return addresses and the instruction pointer).

Having said this, an automata-based program can be considered as a special case of an imperative program, in which implicit part of the state is minimized. The state of the whole program taken at the two distinct moments of entering the *step* code section can differ in the automaton state only. This simplifies the analysis of the program.

**Object-oriented programming relationship**

In the theory of [object-oriented programming](http://en.wikipedia.org/wiki/Object-oriented_programming) an **object** is said to have an internal *state* and is capable of *receiving messages*, *responding* to them, *sending* messages to other objects and changing the internal state during message handling. In more practical terminology, *to call an object's method* is considered the same as *to send a message to the object*.

Thus, on the one hand, objects from object-oriented programming can be considered as automata (or models of automata) whose *state* is the combination of internal fields, and one or more methods are considered to be the *step*. Such methods must not call each other nor themselves, neither directly nor indirectly, otherwise the object can not be considered to be implemented in an automata-based manner.

On the other hand, it is obvious that *object* is good for implementing a model of an automaton. When the automata-based approach is used within an object-oriented language, an automaton model is usually implemented by a class, the *state* is represented with internal (private) fields of the class, and the *step* is implemented as a method; such a method is usually the only non-constant public method of the class (besides constructors and destructors). Other public methods could query the state but don't change it. All the secondary methods (such as particular state handlers) are usually hidden within the private part of the class.