Computer Science • Vol. 8 Special Edition • 2007

Waldemar Grabski*, Michał Nowacki*

CODE GENERATION FOR CSM/ECSM MODELS IN COSMA ENVIRONMENT

The COSMA software environment, developed in the Institute of Computer Science, WUT, was designed primarily for model checking of reactive systems specified in terms of Concurrent State Machines (CSM). However, COSMA supports also Extended CSM (ECSM). The extensions allow for using complex data types and pieces of C/C++ code, attributed to CSM states and/or transitions. Because of these extensions, ECSM models cannot be verified by model checking, but they can be used as an intermediate step in code generation. The underlying CSM represent then the flow of control within cooperating components and the communication among them while the extensions specify the data structures and the details of their processing.

The paper discusses the code generation from ECSM diagrams. The approach is illustrated with an example.

Keywords: *Model Checking, COSMA, Code Generation, CSM, ECSM, FSM*

GENERACJA KODU PROGRAMU NA PODSTAWIE MODELU CSM/ECSM W ŚRODOWISKU COSMA

Środowisko COSMA, rozwijane w Instutycie Informatyki Politechniki Warszawskiej, powstało z myślą o weryfikacji modeli (model checking) systemów reaktywnych specyfikowanych przy pomocy automatów CSM (Concurrent State Machines) jak i ich rozszerzonej wersji (ECSM – Extended CSM). Rozszerzenie CSM o złożone struktury danych, atrybuty związane z przejściami i stanami oraz możliwość bezpośredniego użycia kody w C/C++ powodują, że model wyrażony w ECSM nie może być formalnie weryfikowany, a jedynie stanowić krok pośredni przy generacji kodu. W takim podejściu model CSM reprezentuje sterowanie i komunikację pomiędzy modułami systemu, podczas gdy ECSM – dane i szczegóły przetwarzania.

Artykuł omawia generację kodu z modelu ECSM zilustrowaną przykładem.

Słowa kluczowe: *weryfikacja, COSMA, generacja kodu, CSM, ECSM, automaty skończenie stanowe*

* Institute of Computer Science, Warsaw University of Technology, Warszawa, Poland, W.Grabski@ii.pw.edu.pl, M.Nowacki@ii.pw.edu.pl

1. Introduction

This paper presents a part of the research on the software environment which would combine the UML behavioral specification, formal verification by model checking and code generation. Presently, the commercial design tools that support Unified Modeling Language (UML, e.g. [1]), hardly allow for the formal verification of the model. The UML standards are still evolving, the new ideas appear and research is done to support the designers with formal verification ([2, 3]) and code generation (e.g. HUGO [4]). In the Institute of Computer Science, WUT, the COSMA [5] software environment is used for these purposes. COSMA was primarily designed for model checking. It is conceptually built upon a finite state model called Concurrent State Machines (CSM,[6]) with extensions for complex data types and state actions in Extended CSM(ECSM, see [7]). The general idea is to convert UML behavioral specifications (in terms of state, sequence, cooperation diagrams) into CSM and ECSM models and then to use model checking capabilities of COSMA for the formal verification of selected safety/liveness properties as well as ECSM models for system simulation and code generation.

For each module of the system, the control flow is specified as a finite state Concurrent State Machine, resembling unstructured (and formalized) UML state diagram. The well-defined operation of the product of CSM provides the reachability graph of a system, which can be model-checked against the safety or liveness properties, specified either in terms of temporal formulas or an "observing" automaton. This way, the coordination between communicating / cooperating modules of the system can be formally verified. The ECSM extensions allow to attribute pieces of C/C++ code to states or transitions of CSM models. In contrast to CSM, the ECSM model can only be informally verified by execution or by means of the simulation.

The generation of code from ECSM models has to make use of the information on communication and coordination among processes (which is determined by states and labeled transitions of a CSM) and of data structures and the details of the processing added up in a form of direct C/C++ pieces of code. In our methodology the generated code is treated as a template which should be extended (for example we check intermodule communication protocol without modeling parts of the system which produces and consumes data). The next step in the life-cycle of the system – debugging – also requires human interaction with the code. That's why, in our opinion, it is very important to generate the code which is *human-friendly*.

Recent addition to the COSMA, new tool called WSGenerator, allows to generate the program code from finite state machines model expressed by CSM/ ECSM automata. The preliminary version of WSGenerator [8] was implemented as part of the thesis by Łukasiuk [9]. In this paper we will present basic information about CSM and ECSM, the splitting of the information between CSM and C/C++ code. The simple example of the ATM-Bank system (as in [10]) and the generated code is given.

2. CSM/ECSM model construction rules

The CSM machine is a finite state automaton consisting of states and labeled transitions between them. One of the states is an initial state. States can emit sets of signals. Transitions are labeled with Boolean formulas which represent conditions when a given transition is enabled and may be executed. These conditions depend on signals generated in other automata or received from the environment. Operators +, *, ! stand for Boolean sum, product and complement (respectively). Transitions labeled with Boolean **1** are spontaneous ones, as they are unconditionally enabled. An example of a CSM automaton is shown in Figure 1 (right). A state is represented as rounded rectangle, with state name in the upper part (for e.g. $RetF$) and list of signals emitted in this state – in lower part (e.g. ret_F for $RetF$ state). The transition from the state $IdleF$ to the state $S4$ is enabled when condition $call\_F$ is true (signa $call\_F$ is emitted). A special loop transition is used for remaining in a state (for e.g. the label $!call\_F$ on transition for the state $IdleF$, which means absence of the signal $call\_F$).

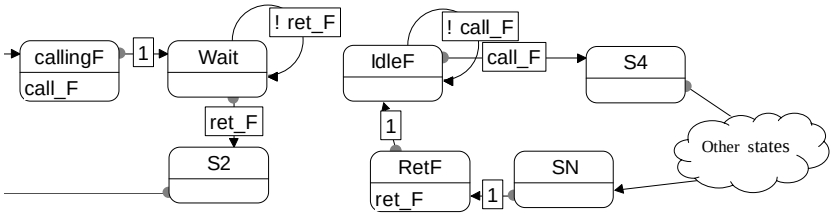

Fig. 1. CSM representation for function call: caller(left) and function(right)

ECSM is an extension of CSM which adds C/C++ actions to the states and C/C++ conditions to the transitions. At present, designer of the system creates the CSM model himself, but work on generating it from the UML is in progress. To allow generation of the *human-friendly* code we additionally have to restrict structure of the automaton (these restrictions will also define the allowed structures in the UML model). In this chapter we describe the construction rules for the common programming structures.

2.1. Calling function

In modeled system every function is represented by a single CSM/ECSM automaton. The call to this function can be modeled in any other automaton in the system. The mechanism that represents function call uses two signals: $call\_xxx$ and ret_xxx where xxx is the function name. The automaton which models calling the function first sends signal which represents function call (in figure Fig. 1 signal $call\_F$), and in the next state it waits for signal which represents return from the function (ret_F signal). The automaton which models the function body has a form of the loop. It waits for the call signal after which the execution of the function starts. Last state in the loop represents the return from the function. After this the automaton returns to idle state.

Function parameters and result are transmitted as ECSM signal's attributes (which form a table of pointers attributed to particular signals). For a function with one parameter of type short the following ECSM code will be added for sending *call_xxx* : `call_xxx[0] = (int)(short*)&value;`, and on receiving side – for response to signal *call_xxx* : `param = *(short*)(int)call_xxx[0];`. Function body can only contain state structures which reflect constructions in structural programming languages. As an instruction we understand: basic, conditional or while loop instruction and the sequence of instructions.

2.2. Basic instruction

The basic instruction is modeled by a single state as shown in Figure 2. The ECSM action in this state can contain piece of code which will be inserted into final program.

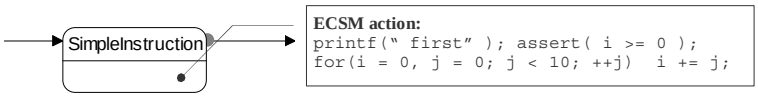

Fig. 2. Simple Instruction

Of course this is the basic instruction from the model point of view. The ECSM action can contain complex C/C++ code.

Example: code generated from automaton in Figure 2:

```
printf("first");
assert( i >= 0 );
for(i = 0, j = 0; j < 10; ++j)
   i += j;
```

2.3. Conditional execution

The model of the conditional execution contains an initial and end state. The name of the end state must be the same as the initial state's prefixed with "End". All sequences from initial state to end state describe alternative executions (sequences of instructions as described below). The end state has to be direct successor of the initial state. If in all cases some code should be executed, additional transition between these states has to be added with condition set to false. Figure 3 presents a model of conditional execution with simple instructions on each path (that's why there is an arc from *If* to *EndIf* with always false condition).

Example: code generated from the automaton in Figure 3:

```
if( x > 0 ){
   j = 1;
}
else{
   j = 3;
}
```

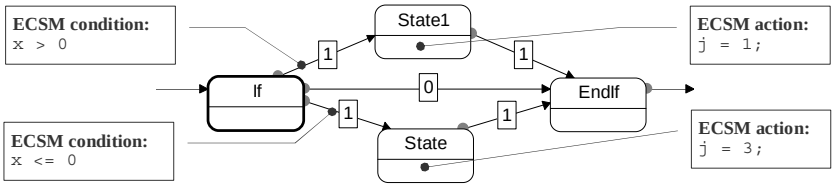

Fig. 3. Conditional instruction

2.4. While loop instruction

The model of the while loop contains the initial state and the end state. The name of the end state must be the same as the initial state prefixed with "End". The sequence of instructions form a loop which begins and ends in initial state. The loop conditions are placed as an ECSM condition for transitions outgoing from initial state. These two conditions must be complementary. One of them is used in generated code as a condition for a loop. Automaton which models a while loop with two simple instructions is shown in Figure 4.

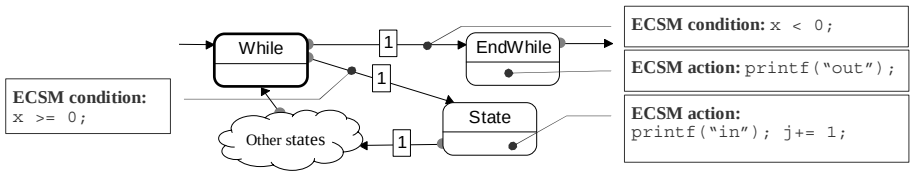

Fig. 4. While loop instruction

Example: code generated for automaton in Figure 4:

```
while( x >= 0 ){
   printf("in"); j += 1;
}
printf("out");
```

2.5. Instructions sequence

The instructions sequence is described as a sequence of instructions with a single successor. Each of the sequence elements can be a simple or complex (conditional, loop or the sequence of instructions) instruction. CSM and ECSM conditions on transitions between the end of instruction and next instruction are always true.

3. Example: ATM-BANK system

The methodology described above will be illustrated in the ATM-Bank example as in [10], except the concurrency in the Bank state diagram (we use the UML diagrams as they are most universal). The bank system is simplified and it contains only one

ATM and only one card can be used. Cooperation between Bank and ATM consists only of the validation of PIN and the validation of the card. The ATM class contains no variables, all the information about the card is stored in the bank. In this system there is only one card, so the information about it is represented as three variables (`cardValid, numIncorrect, maxNumberIncorrect`) for representig the state of the card.

The automatic teller machine communicates with the bank to validate PIN. The bank can answer to this request in three ways. If both card and PIN are valid then bank sends the information that PIN is valid. In second scenario – card is valid, but PIN is incorrect – bank sends request to re-enter PIN. This system allows only three incorrect PIN entries, after this the card is blocked. The third scenario is that the bank sends the information that the card is invalid. In case of correct PIN, the ATM dispenses the money, in case of incorrect PIN – allows PIN to be reentered twice, when card is invalid – refuses to use the card and the card is returned. ATM's behavior is presented as UML state diagram (Fig. 5).

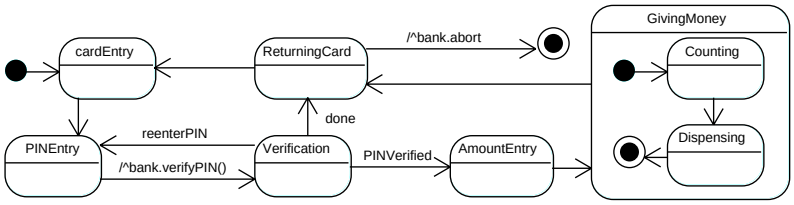

Fig. 5. ATM state diagram

No information about PIN entries is stored in ATM during one session or between different sessions with bank. This information is stored in the Bank and is used when ATM requests PIN checking. If card owner exceeds the number of allowed PIN errors the card will be blocked and the system shouldn't allow to withdraw money with this card. The `cardValid`, `numberIncorrect` and `maxNumberIncorrect` attributes in the bank class are used to implement this requirements. State diagram for the Bank is presented in Figure 6.

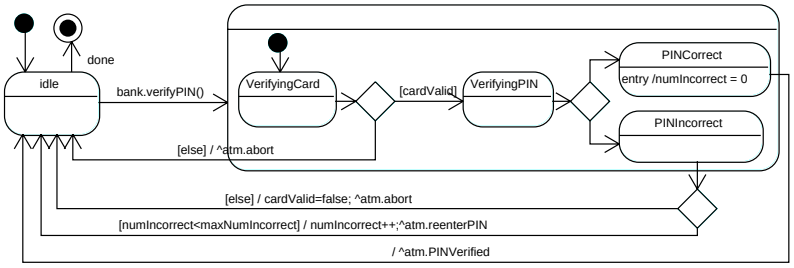

Fig. 6. Bank state diagram

4. ECSM model of the bank system

Our model consists of two parts: ATM and Bank. ATM is represented as a single function (atm) modeled by a single automaton (Fig. 7). Bank is represented as a group of functions, where one (bankVerifyPIN) is called from ATM, while the other ones (checkCard, checkPIN) are internal. In all cases the CSM level models function calls and program structure, while ECSM – system logic actions and conditions. In this paper, for Bank part, we present only CSM/ECSM automata for *bankVerifyPIN* and *checkPIN*.

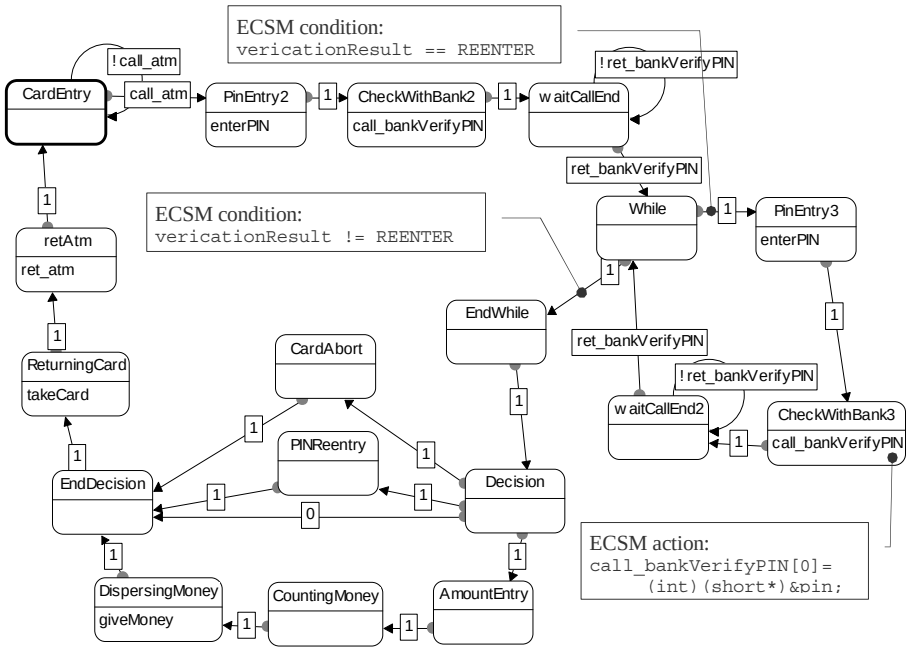

Fig. 7. ATM's automaton model (with some of the ECSM actions and conditions)

Each function is represented by a single CSM/ECSM automaton. Because of the similarities in these functions and mechanism for calling them, we present only *checkPIN* (Fig. 8) and *bankVerifyPIN* (Fig. 9) functions.

In the automaton representation of the Bank's main function (Fig. 9) only some of the ECSM's conditions and actions were shown.

The remaining are:

- the ECSM conditions based on the value of numIncorrect for outgoing transitions from PINInvalid state,
- the ECSM action `returnValue = ABORT;` in CardInvalid state,
- the ECSM actions for function calls: *checkCard* and *checkPIN*.

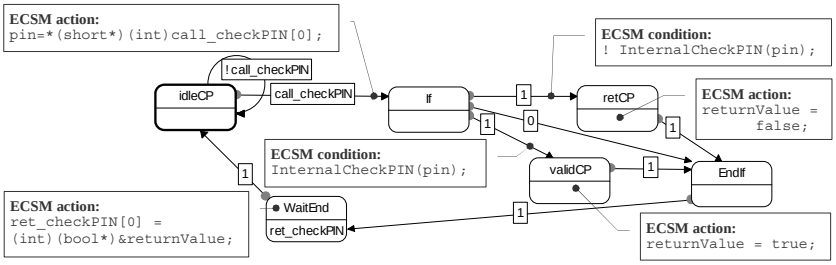

Fig. 8. Automaton model of $checkPIN$ function

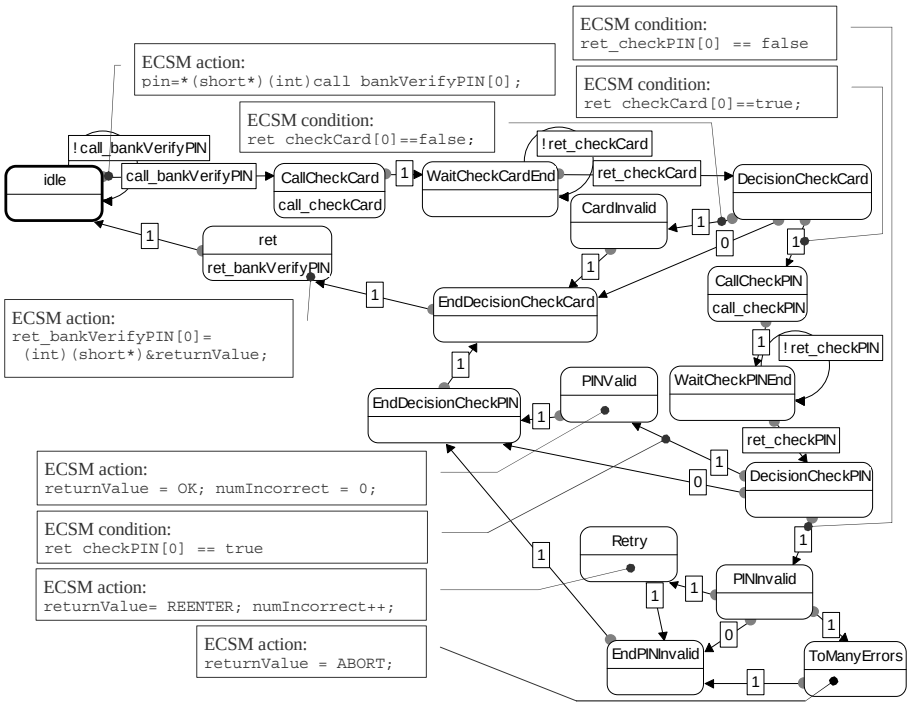

Fig. 9. Bank's main function ($bankVerifyPIN$)

4.1. The code generated from CSM/ECSM Model

Below, the generated code for presented part of the model is shown[1].

[1] The presented code is formated by hand to save space by adjusting white spaces. The CSM/ECSM ATM-BANK model, unedited generated code and WSGenerator tool is available on `http://www.ii.pw.edu.pl/cosma/code_generator.htm`.

This code was created by the new Cosma module (WSGenerator).

```
enum VerifyPINReturnType{ ABORT, REENTER, OK };

void atm(){
   short pin;   pin=0;
   VerifyPINReturnType verificationResult;
   verificationResult = ABORT;
   pin = InternalGetPIN();
   vericationResult = bankVerifyPIN(pin);
   while (vericationResult == REENTER)
       {   pin=InternGetPIN();
           vericationResult=bankVerifyPIN(pin);}
   if (vericationResult == ABORT) { }
   else
      if (vericationResult==OK)
         {printf("Take money.");}
      else{ assert( 0 && "Impossible situation" ); }
  printf( "Take card." );
}

VerifyPINReturnType bankVerifyPIN(short pin){
   bool pinValid;            bool cardValid;
   short maxNumIncorrect;    short numIncorrect;
   VerifyPINReturnType returnValue;
   pinValid = false;         cardValid = false;
   maxNumIncorrect = 2;      numIncorrect = 0;
   returnValue = ABORT;

   cardValid = checkCard();
   if (!cardValid){
      returnValue = ABORT;
   }else{
      pinValid = checkPIN(pin);
      if (pinValid){
         returnValue=OK;
         numIncorrect=0;
      }else{
         if (numIncorrect <= maxNumIncorrect){
            numIncorrect ++;
            returnValue = REENTER;
         }else{   returnValue = ABORT;   }
      }
   }
```

```
   return returnValue;
}

bool checkPIN(short pin){
   bool returnValue;
   returnValue = false;
   if (! InternalCheckPIN(pin))
      { returnValue = false; }
   else
      { returnValue = true; }
   return returnValue;
}
```

5. Conclusions

The quality of the created system strongly depends on quality of design, so the support for the designer, from the very beginning of the design process is very important. It can be achieved with the tools for:

- creating the description of the system in clear and unambiguous way,
- formal verification of the model,
- automatic code generation.

We have shown the approach to the automatic code generation, which produces program code in the form that simplifies the programmer's work in the debugging and extending the application, while the communication skeleton of the system (specified in terms of CSM) can be still formally model-checked. One can hope that the transformation of UML state diagrams into CSM/ECSM will be added to COSMA environment soon so that the results described above will be used also for the generation of the code from this commonly known specification.

References

[1] Unified Modeling Language: `http://www.omg.org/technology/documents/formal/uml.htm`

[2] Berard B. (ed.) et al.: Systems and Software Verification: Model-Checking Techniques and Tools, Springer Verlag, 2001

[3] Clarke E. M., Grumberg O., Peled D. A.: *Model Checking*, MIT Press, 2000

[4] Project Hugo: `http://www.pst.informatik.uni-muenchen.de/projekte/hugo/`

[5] COSMA project: `http://www.ii.pw.edu.pl/cosma/`

[6] Mieścicki J.: Concurrent StateMachines, the formal framework for model-checkable systems, ICS Research Report 5/2003

[7] Krystosik A.: *ECSM — Extended Concurrent State Machines*. ICS Research Report 2/2003

[8] WSGenerator tool: `http://www.ii.pw.edu.pl/cosma/code_generator/`

[9] Łukasiuk K.: *Construction of Internet applications based on client—server architecture using COSMA design environment*. M.Sc. diploma ISC, 2006 (in Polish)

[10] Knapp A., Merz S.: *Model Checking and Code Generation for UML State Machines and Collaborations*, Proc. 5th Wsh. Tools for System Design and Verification 2002, p. 59–64