AUTOMATIC CODE GENERATION BASED ON FORMALLY ANALYZED UML STATECHART MODELS

Gergely Pintér, István Majzik
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Address: Magyar tudósok körútja 2., Budapest, Hungary, H-1111
Phone: + (36-1) 463-3582, Fax: + (36-1) 463-2667, E-mail: pinterg@mit.bme.hu, majzik@mit.bme.hu

Abstract: This paper aims at providing an efficient implementation pattern for source code level instantiation of UML statcharts. The code generation is based on extended hierarchical automata, the formal description method used as an intermediate representation of statecharts for model checking purposes, this way enabling automatic implementation of formally analyzed models.

Since statecharts can automatically be mapped to extended hierarchical automata, a code generator based on our pattern could be used as a module that can be inserted into any UML modeling tool equipped with model export capabilities. This approach enables the modeler to use the usual design environment and hides the transformation required for model checking and code generation steps.
Keywords: UML, statechart, Extended Hierarchical Automaton, code generation.

1. INTRODUCTION

Applying computer-based systems in safety-critical areas poses high reliability requirements against their software. Programs running on these systems must be proven correct i.e. free from design and implementation faults. Elimination of design faults is addressed by model checking, possibility of inducing errors in the implementation and maintenance phase can be reduced by automatic code generation.

The focus of this article is the control core of event-driven embedded computer systems. These applications can be specified by finite state-transitions systems. A state of the system represents a situation during which some invariant condition holds. The system responds to external stimuli with actions and transitions in the state space. There are several formal and non-formal facilities for describing the behavior of such systems. One of the most popular of them is the statechart package of the Unified Modeling Language (UML) (OMG, 2001).

Most of UML modeling environments aim at only providing a visual design environment, their code generation capabilities are modest, they can implement typically the static parts of the software (class headers, function prototypes etc.) barely, only the most advanced tools (I-Logix Rhapsody, IAR Visual State) are capable of generating the code of the control core (i.e. event dispatcher, state machine implementation etc.). Although these systems provide facilities for investigating the response of the system to external stimuli in a visual (statechart-level) debugging environment, code generation based on formally analyzed models remains an open issue.

UML is a semi-formal language, the syntax and static semantics is defined precisely but the dynamic semantics is defined only informally.

A formal operational semantics of UML statecharts is presented in Latella *et. al.* 1999b. In that approach statechart diagrams are first mapped to the intermediate format of extended hierarchical automata (EHA) and the formal semantics is defined for these automata based on Kripke structures. This formal behavioral specification forms the basis of automatic verification of the model by using the SPIN model checker as described in Latella *et. al.* 1999a. In this latter approach the EHA model is translated into PROMELA, the specification language of SPIN.

The goal of this paper is providing an efficient implementation pattern for source code level instantiation of the extended hierarchical automata, the formal description language used as an intermediate representation of UML statecharts, therefore presenting a way for automatic code generation based on formally analyzed models. Since statecharts can automatically be mapped to extended hierarchical automata, a code generator based on this pattern could be

used as a module that can be inserted into any UML modeling tool equipped with model export capabilities (i.e. implementing the XML Metadata Interchange (XMI) format). This approach enables the modeler to use the usual design environment and hides the transformation required for model checking and code generation steps.

The idea of the suggested pattern is quite simple: (i) provide a compact representation (data model) of the extended hierarchical automaton and (ii) implement an efficient "interpreter" for this model. The interpreter can be described as a programming language level representation of the PROMELA code as generated in Latella *et. al.* 1999a in a generalized form. The hardware requirements of the pattern are low, therefore applicable even in small embedded systems.

Although the idea is quite simple, the code generation this way is not only a next transformation step from extended hierarchical automata to a programming language. Since the original EHA model does not represent some parts of the UML statechart concepts extensions are proposed to enable the implementation of state entry and exit actions. These extensions are required by the code generation only, their introduction does not interfere with the mathematical model therefore the analysis results achieved on the original model apply to the extended one as well.

An introduction to the syntax and semantics of UML statecharts and extended hierarchical automata is presented in Sect. **2**, the suggested pattern is outlined in Sect. **3**. Finally conclusions are drawn in Sect. **4**.

2. ABSTRACT MODELS

This section introduces UML statecharts (OMG 2001) the description methods used for high-level modeling and extended hierarchical automata (Latella *et. al.* 1999b) the abstract model for mathematical analysis. Both introductions define the *syntax* first then outline the *operational semantics*.

2.1 UML Statecharts

The State Machine package of UML specifies a set of concepts that can be used for modeling discrete behavior through finite state-transition systems.

The *syntax* is well defined by the metamodel (i.e. a class diagram of the model) of the statechart package in the standard (OMG 2001). Besides the fundamental building elements (states and transitions) it provides several sophisticated constructs that make the high-level description of the control flow easier:

- *States* model situations during which some invariant condition holds. Optional entry and exit actions can be associated to them to be performed whenever the state is entered or exited.
- *Transitions* are directed relationships between a source and a target state. An optional action can be associated to them to be performed when firing the transition.
- States can be refined into *substates* resulting in a state hierarchy. The decomposition can be simple refinement (only one of the substates is active at the same time) or orthogonal division where all the substates (called regions) are active at the same time. Join and fork vertices can be used to represent transitions originating from or ending in states in different orthogonal regions. Transitions are allowed to cross hierarchy levels.
- *Shallow* and *deep history* pseudostates are available as shorthand notations to represent the most recent active substate or configuration of the containing composite state.
- Transitions can be *guarded* by boolean expressions that are evaluated when an event instance is dispatched by the state machine. If the guard is true at that time, the transition is enabled, otherwise it is disabled.

The *operational semantics* is expressed only informally in the standard.

The execution of state machines is driven by *events*. A transition is *enabled* if all of its source states are active, the event satisfies its trigger and its guard is enabled. Two transitions are in *conflict* if the intersection of the states they exit is non-empty. *Priority* of transition t_1 is higher than the priority of t_2 if the source state of t_1 is a directly or transitively nested substate of the source state of t_2.

After receiving an event a maximal set of enabled transitions is selected that are not in conflict with each other and there is no enabled transition outside the set with higher priority than a transition in the set. The transitions selected this way fire in an unspecified order.

The exact sequence of actions to be performed when taking a transition is specified by the standard: first the exit actions of all states left by the transition are executed starting with the

deepest one in the hierarchy, next the action associated to the transition is performed finally the entry actions of states entered by the transition are executed starting with the highest one in the hierarchy.

The *example* discussed in this article is the traffic supervisor system in the crossing of a main road and a country road. The equipment at the main road consists of two sensors, a camera, a traffic light and the controller. For simplicity reasons only the light control of the main road is investigated here. The first sensor signals the arrival of a car to the crossing from the main road and the second one sends a signal to the controller if a car runs in the crossing from the main road.

The controller system provides higher precedence to the main road i.e. it does not wait until the normal time of switching from red to red-yellow if more than two cars are waiting at the main road but switches immediately (the arrival of a car to the crossing is indicated by one of the sensors).

Cars running illegally in the crossing during the red signal are detected by the second sensor and can be recorded by the camera connected to the controller. The camera can be switched on and off manually.

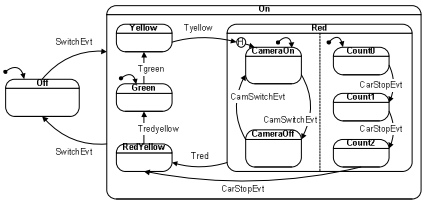

Fig. 1. UML statechart of the traffic light example

The statechart diagram (*Figure 1*) shows only the behavioral specification of the light control at the main road. The light can be operating (*on* state) or switched off (*off* state). The transition between them is triggered by the *switch* event.

Representation of light states (*red, red-yellow, green* and *yellow*) is obvious, transitions between these states are triggered by time events. State *red* is decomposed orthogonally into two regions, one for the camera and one for counting the cars waiting at the main road. The most recent state of the camera (operating or switched off) is restored when entering the *red* state through the history pseudostate. The interlevel transition from *count2* to *red-yellow* is triggered by the arrival of the third car from the main road. This way the crossing is provided to the traffic from the main road.

2.2 Extended Hierarchical Automata

The syntactic transformation from UML statecharts to extended hierarchical automata (EHA) aims at the formalization of the specification and separation of concepts obscured in the UML model (blurred representation of hierarchy and concurrency, interlevel transitions, etc.). It provides a clear and mathematically analyzable model of state refinement, concurrency and transitions.

The *syntax* of extended hierarchical automata is described in a functional notation in Latella *et. al.* 1999b. In the following a short informal overview is given mainly concentrating on the representation of UML concepts. For explained definition refer to Latella *et. al.* 1999a and Latella *et. al.* 1999b.

An EHA consists of *sequential automata*. A sequential automaton contains simple (non-composite) *states* and *transitions* between them. These states represent simple and composite states of the UML model. States can be *refined* to any number of sequential automata.

All the refinement automata of a state are running concurrently, this way UML concurrent composite states can be modeled by EHA states refined to several automata representing one region each. A non-concurrent composite state is refined to only one automaton.

Transitions may not cross hierarchy levels (i.e. their source and target state are in the same automaton). Interlevel transitions of the UML model are substituted by labeled transitions in the automata representing the lowest composite state that contains all the explicit source and target states of the original transition. The labels are called *source restriction* and *target determination*. The source restriction set contains the original source states of the transition in the UML statechart while the target determination set enumerates the original target states. Both sets contain states of the (possibly transitively) refinement automata of the source or the target state.

The *operational semantics* is expressed by a Kripke-structure in Latella *et. al.* 1999b.

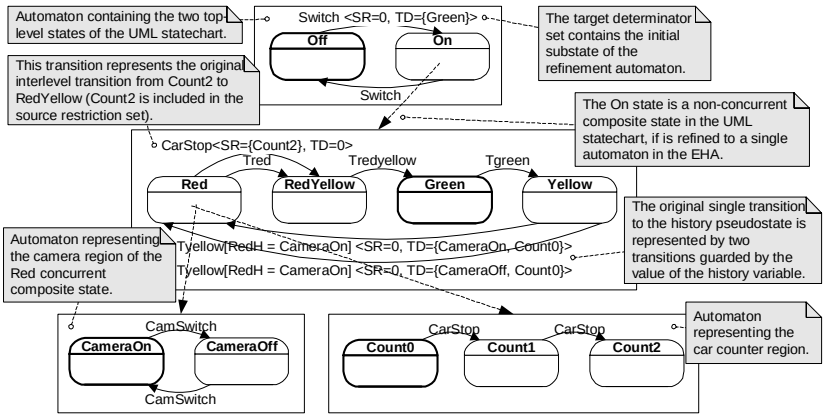

Fig. 2. EHA representation of the traffic light example

The execution of extended hierarchical automata is driven by *events*. A transition is *enabled* if its source state and all states in the source restriction set are active, the actual event satisfies the trigger and the guard is enabled. *Priority* of transition t_1 is higher than the priority of t_2 if the original source state of t_1 in the UML model is a directly or transitively nested substate of the original source of t_2. The original source of a transition is indicated by the source restriction set associated to the transition.

An enabled transition can fire if there are no transitions enabled with higher priority. On taking the transition the target state and all states in the target determination set are entered.

The extended hierarchical automaton of the traffic light *example* (*Figure 2*) represents the UML statechart in a clear refinement hierarchy. Statechart states (simple and composite ones) are mapped to EHA states (*on, off, red, red-yellow, green, yellow, camera-on, camera-off, count0, count1* and *count2*). Concurrent and non-concurrent refinement is expressed by automata assigned to states. This way the non-concurrent *on* state is refined to a single automaton containing the *red, red-yellow, green* and *yellow* states while the concurrent composite state *red* is refined to two automata representing one region each. Note that automata can represent the internal structure of a composite state and a region of a concurrent composite state as well.

Since the history state in the camera region of the *red* state cannot be expressed directly in EHA it was substituted by transitions targeting the states possibly pointed by the history state (*camera-on* and *camera-off*) guarded by the value of a "history variable" (`RedH`). Note that this substitution aims at explaining a shorthand notation barely, the resulting model remains to be a well-formed UML statechart therefore this preprocessing does not interfere with the formal analysis possibly performed on the EHA representation.

The original EHA model used by Latella *et. al.* 1999a and Latella *et. al.* 1999b does not deal with entry and exit actions since these features do not belong to the mathematical abstraction. In order to enable the representation of state entry and exit actions an extended form of the EHA metamodel (Varró *et. al.* 2002) was developed that forms the basis of the data model used by the implementation pattern outlined in the following section. If entry and exit actions do not generate new events their introduction does not modify the mathematical model.

Entry and exit actions are associated to the `State` metaclass in the UML metamodel and these associations are inherited by the descendant simple and composite states, final states etc. (regions of a concurrent composite state are composite states themselves as well).

Since simple and composite states that are not regions in the UML model are converted to states of the EHA and regions are represented by automata, entry and exit actions has to be associated not only to EHA states but to the automata as well (automata that do not represent regions should have empty entry and exit actions). This modification can be modeled by adding the `StateRepresentation` abstract metaclass

to the EHA metamodel as ancestor of `State` and `Automaton` metaclasses and two associations from it targeting the `Action` metaclass with role names `entryAction` and `exitAction` (*Figure 3*).

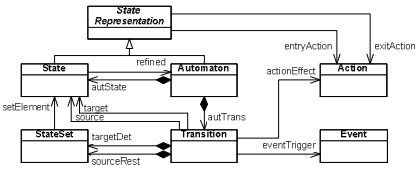

Fig. 3. The EHA metamodel with extensions representing entry and exit actions

2.3 Model checking based on extended hierarchical automata

Extended hierarchical automata are used as intermediate representation for model checking (Pap *et. al.* 2001). Several properties can be investigated like safety (avoidance of hazardous states), liveliness (reachability of proper states) and determinism. The formal analysis takes place before the implementation in the early phases of the development. The model that is proven to be correct is ready to be implemented.

3. THE IMPLEMENTATION PATTERN

This section introduces the implementation pattern developed for instantiating extended hierarchical automata. Since we would like to present a solution that is usable in systems equipped with small memory, the in-core representation of the EHA must be efficient in terms of memory consumption and processing power requirements.

Several common approaches were taken into consideration while designing the implementation pattern like the object-oriented *State* design pattern (Gamma *et. al.* 1994), the Quantum Hierarchical State Machine pattern (Samek *et. al.* 2002) and other widely used solutions. However none of them provided support for handling the concurrency.

The pattern proposed here can be divided into three parts: the expression of the *static structure* (state—automaton hierarchy), a bit pattern for storing the *configuration* (active states) and the *interpreter* that takes a static structure, a configuration and an event and performs the necessary actions and updates the configuration.

As it will be seen the static structure is a modified, extended and preprocessed form of the original EHA metamodel and the interpreter performs corresponds to the PROMELA representation without implementing non-deterministic behavior.

3.1 Static structure and configuration information

The separation of the static structure and the actual configuration information reduces the memory consumption since in an application that consists of several instances of a class described by an EHA only one instance of the static structure description is needed. The configuration of the instances can be expressed with bit vectors.

The static structure (*Figure 4*) is a modified and preprocessed form of the EHA metamodel (*Figure 3*). Modifications were taken to enable the faster navigation and smaller memory consumption.

The topmost container of the static information is the `Structure` class. Its association targeting automata, states and transitions are stereotyped with *ordered*, showing that these associations must be implemented by containers that preserve the order of elements (e.g. an array), thus position of the objects in the list (e.g. the array index) can be used as unique *identifier* amongst objects of the same class (transitions, states etc.).

The containing relation between automata and states, the state refinement, transition triggers and guards, entry and exit actions and actions associated to transitions are represented by associations in the obvious way.

The operating rules are described by associations originating from the `Transition` class. States that must be active to enable the transition (the source restriction set and the source of the transition) are collected in an association with the *enabling* role. States to be entered when taking the transition are collected in an other association with the *entered* role. These associations are marked with the stereotype *ordered* as well. This way the order of states to be entered when taking a transition can be pre-calculated and stored. Representing the source of the transition does not need a separate association, since it can be stored in the first element of the ordered *enabling* set by convention.

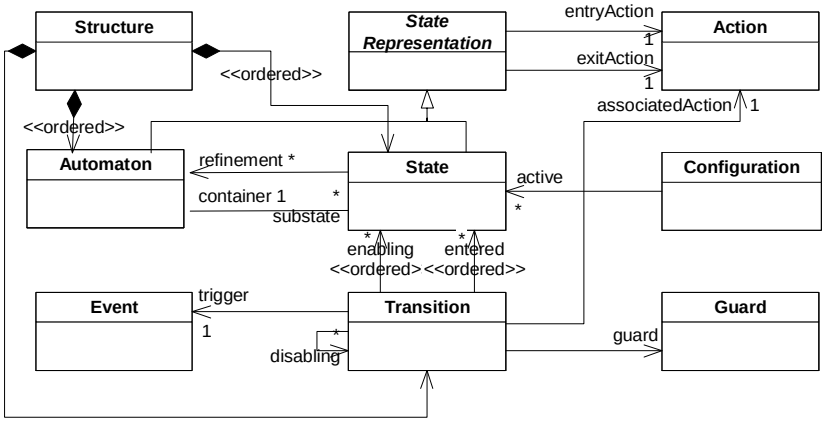

Fig. 4. Class diagram of the proposed pattern

Transitions that have higher priority than the actual one are collected in the *disabling* set. The transition is fireable only if none of these transitions are enabled.

The class structure can be effectively implemented in ANSI C. `State`, `Transition` and `Structure` classes can be represented by C structures, events and guards can be implemented as functions. Associations targeting functions (`Action`} and `Guard` classes) can be function pointers while other associations can be implemented by storing the identifier of the pointed object. Storing IDs instead of pointers can greatly reduce the memory requirements since an identifier can be much shorter than a pointer (e.g. if there are no more than 256 states, transitions and automata then the identifiers can be bytes that are four times shorter than the memory addresses in a 32 bit architecture).

The actual configuration (active states) is represented by the `Configuration` class. The association targeting the `State` class can be implemented by an ordered bit vector (i.e. i^{th} element of the vector is true if and only if the state with ID i is active) providing this way an extremely compact representation of the object state. Note that only this bit vector should be stored for each instances of the class described by the statechart, the static representation is read-only therefore can be stored in a single instance.

3.2 Dynamic behavior

The interpreter function is based on the formal semantics (PROMELA code) as described in Latella *et. al.* 1999a. In that approach a step of the process consists of the following phases:

1. Selection one of the available events. The storage method of events (FIFO, LIFO, set, multiset etc.) is not fixed by the model.
2. Selection of enabled transitions. A transition is enabled if and only if its source state and all states in the source restriction set are active, the selected event is the trigger and the associated guard evaluates to true.
3. Conflict resolution based on priority relations (i.e. selection of “fireable” transitions). A transition is fireable if there are no enabled transitions with higher priority.
4. Non-deterministic selection of a maximal set of fireable transitions.
5. Firing the selected transitions (calculating the resulting state configuration and performing actions associated to the transition).

The programming language level representation follows the same algorithm with minor modifications. The interpreter can be implemented as a function parametrized by the static structure description, the actual configuration and the event to be dispatched (therefore the event selection method is out of the scope of the interpreter).

The UML statechart model enables the existence of conflicting transitions even after apply-

ing priority rules (e.g. there are two transitions originating in the same state with the same trigger event and overlapping guards). In these cases the selection of the maximal set of transitions to fire is non deterministic. This obscurity is acceptable in the design phase indicating non-elaborated parts but must be eliminated from the final model especially in case of safety critical systems where non-deterministic behavior can lead to catastrophic consequences. This way the model instantiated by our pattern is required to be free from non-determinism in the sense that all the conflicts amongst transitions must be resolved by priority relations. Thus the non-deterministic selection does not take place, all the fireable transitions fire.

The original model does not deal with entry and exit actions associated to states. These actions are obviously essential in the implementation therefore the interpreter must ensure their execution.

Entering a composite state requires entering one of its substates (in each one of concurrent regions in case of concurrent composite states). Since a composite state can be entered in several ways (default entry, explicit entry into a substate, shallow or deep history, entry through a fork pseudostate or directly into a region etc.) runtime calculation of the states to be entered and the order in entry actions to be called would be very time-consuming. The implementation pattern suggests the pre-calculation of this entry chain and storing it in the ordered association between the `Transition` and `State` class (*entered* role). This solution greatly simplifies the implementation of performing the state entry actions at the cost of a little redundancy in the model. This way the interpreter can simply walk through this (ordered) list and call the entry actions associated to the states and their containing automata in the list.

States to *exit* from cannot be calculated during the code generation since it depends on the actual configuration of the object when receiving the event triggering the transition. As OMG 2001 specifies, when exiting from a composite state its active substate is exited recursively. This means that the exit actions are executed in sequence starting with the innermost active state in the current state configuration. When exiting from a concurrent state, each of its regions are exited. After that, the exit actions of the regions are executed.

The simplest solution for implementing this behavior is a recursive function that traverses the refinement tree and calls the appropriate exit actions of states and their containing automata. This function can be described with the following pseudocode:

```
recursiveExit(State s)

  // r is an automaton, refinement of s
  forEach s.refinement as
    recursiveExit(activeSubstateOf r);

    // Exit action of the automaton
    r.exitAction();

    // Exit action of the state
    s.exitAction();

    markInactive(s);
```

The complete pseudocode of the interpreter function can be described by the following pseudocode:

```
step(Structure str, Configuration cfg,
     Event e)

  // Collect enabled transitions
  enabled = collectEnabled();

  // Collect fireable transitions
  fireable = collectFireable();

  // t is a fireable transition
  forEach fireable as t
    // Recursive exit from the source
    recursiveExit(t.source);

    // Action associated to the transition
    t.associatedAction();

    // The container automaton of the target
    // state is not entered so it is handled
    // separately

    // Entry action of the target state
    t.entered[0].entryAction();

    // Mark the target state active
    markActive(t.entered[0], cfg);

    // s is a state to be entered
    forEach t.entered[1...] as s
      // Entry action of the containing
      // automaton
      s.container.entryAction();

      // Entry action of s
      s.entryAction();

      // Mark the state active
      markActive(s, cfg);
```

Here the pseudo function `collectEnabled` stands for collecting the enabled transitions (i.e. source states are active, the trigger is the actual event and the guard evaluates to true) while `collectFireable` represents the selection of enabled transitions that are not dis-

abled by any other transition with higher priority (*disabling* set).

3.3 Example

The prototype of the interpreter function and the static structure was implemented in C. The memory consumption of the static structure depends on the length of identifiers and the word size of the architecture. In the case of the traffic light example according to our calculations the static description should fit in about 1 kB on a machine with 32 bit long word size when choosing 32 bit long identifiers and should fit in less than half kB on a machine with 16 bit long addresses when choosing 8 bit long identifiers. Since there are 11 states in the model the configuration information of an object fits in 11 bits (two bytes).

4. CONCLUSIONS AND FUTURE WORK

Formal analysis of abstract models addresses the elimination of design faults in the early phases of the development. Automatic code generation based on checked models reduces the possibility of inducing errors in the implementation and maintenance phase.

In this paper an efficient implementation pattern was proposed for source code level instantiation of UML statecharts after transforming them to extended hierarchical automata. Extending the pattern with runtime self checking capabilities and testing support are the subject of our future work.

REFERENCES

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. (1994) Design Patterns Elements of Reusable Object-Oriented Software. Addison Wesley.

Latella, D., I. Majzik, and M. Massink. (1999) Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. In *Formal Aspects of Computing*, volume 11. Springer Verlag, 637-664.

Latella D., I. Majzik, and M. Massink. (1999) Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Proc. FMOODS'99, the Third IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*. Firenze, Italy 331-347.

OMG. (2001) *Unified Modeling Language (UML) Version 1.4*.

Samek, M. (2002) *Practical Statecharts in C/C++*. CMP Books.

Varró, D., G. Varró and A. Pataricza. (2002) Designing the Automatic Transformation of Visual Languages. *Science of Computer Programming*, 44(2):205-227.

Pap, Zs., I. Majzik, and A. Pataricza. (2001) Checking General Safety Criteria on UML Statecharts. In *Lecture Notes in Computer Science*, number 2187. Springer Verlag, 46-55.