Program Code Generation based on UML Statechart Models

Gergely PINTÉR, István MAJZIK
Budapest University of Technology and Economics
Department of Measurement and Information Systems
pinterg@mit.bme.hu, majzik@mit.bme.hu

Abstract

Since visual modeling languages are getting more and more popular, automatic generation of program code on the basis of high-level models is an important issue. This article discusses implementation possibilities of statecharts, the graphical notation for describing state-based event-driven behavior in the Unified Modeling Language (UML). The first part of the article outlines common approaches published in the literature and identifies their weaknesses. In the second part an implementation pattern is proposed that is capable of efficiently instantiating most of the statechart features. The pattern developed by us poses low hardware requirements therefore applicable even in embedded systems.

1 Introduction

This article discusses implementation possibilities of UML (Unified Modeling Language [10]) statecharts. After a short introduction to UML statecharts the common implementation approaches are introduced with emphasis on their hardware requirements and features covered by them. The overview concludes that none of the well-known techniques provide complete solution for implementing concurrent operation, this way a custom solution needs to be developed.

The second part of the article introduces an implementation strategy developed by us based on the transformation from UML statecharts to Extended Hierarchical Automata. The resulting pattern is capable of instantiating the most important features of UML statecharts including state hierarchy and concurrent operation. The modest hardware requirements of the pattern enable its usage even in embedded systems based on low-end microprocessors or microcontrollers.

The problem has emerged in a research targeting implementation possibilities of automatic self checking of program execution. The expected outcome of this project is a code generator extension that is able to create not only the state machine engine (control core) of the application but to implement modules for run-time self checking, to insert methods supporting testing (observers, invariant checks) and to create test cases and harnesses for classic testing as well. This way the UML model serves not only as a formal basis for automatic implementation but as the reference information for error detection as well (Fig. 1).

In this context the design pattern used for code generation provides solution for implementing the control core and enables run-time self-checking and classic testing by providing points of observation of the control flow.

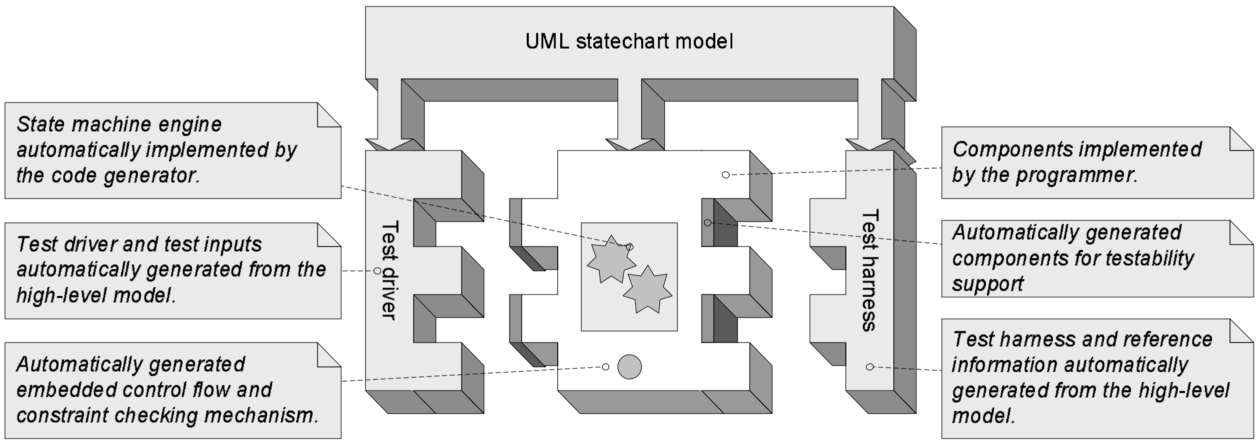

UML statechart model
State machine engine automatically implemented by the code generator.
Test driver and test inputs automatically generated from the high-level model.
Automatically generated embedded control flow and constraint checking mechanism.
Test driver
Test harness
Components implemented by the programmer.
Automatically generated components for testability support
Test harness and reference information automatically generated from the high-level model.

Figure 1: Overview of the context

2 Describing dynamic behavior in UML

This section introduces UML statecharts [10] the description methods used for high-level modeling of program behavior. First we will define the abstract *syntax* then outline the *operational semantics*.

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 precisely defined by the metamodel (i.e. a class diagram describing the model elements) in the standard (Fig. 2). Besides the fundamental building elements (states and transitions) it provides several sophisticated constructs that make the 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 the transition fires.
- 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*. Events are stored in an event queue until they are selected by the dispatcher mechanism. The semantics of event processing is based

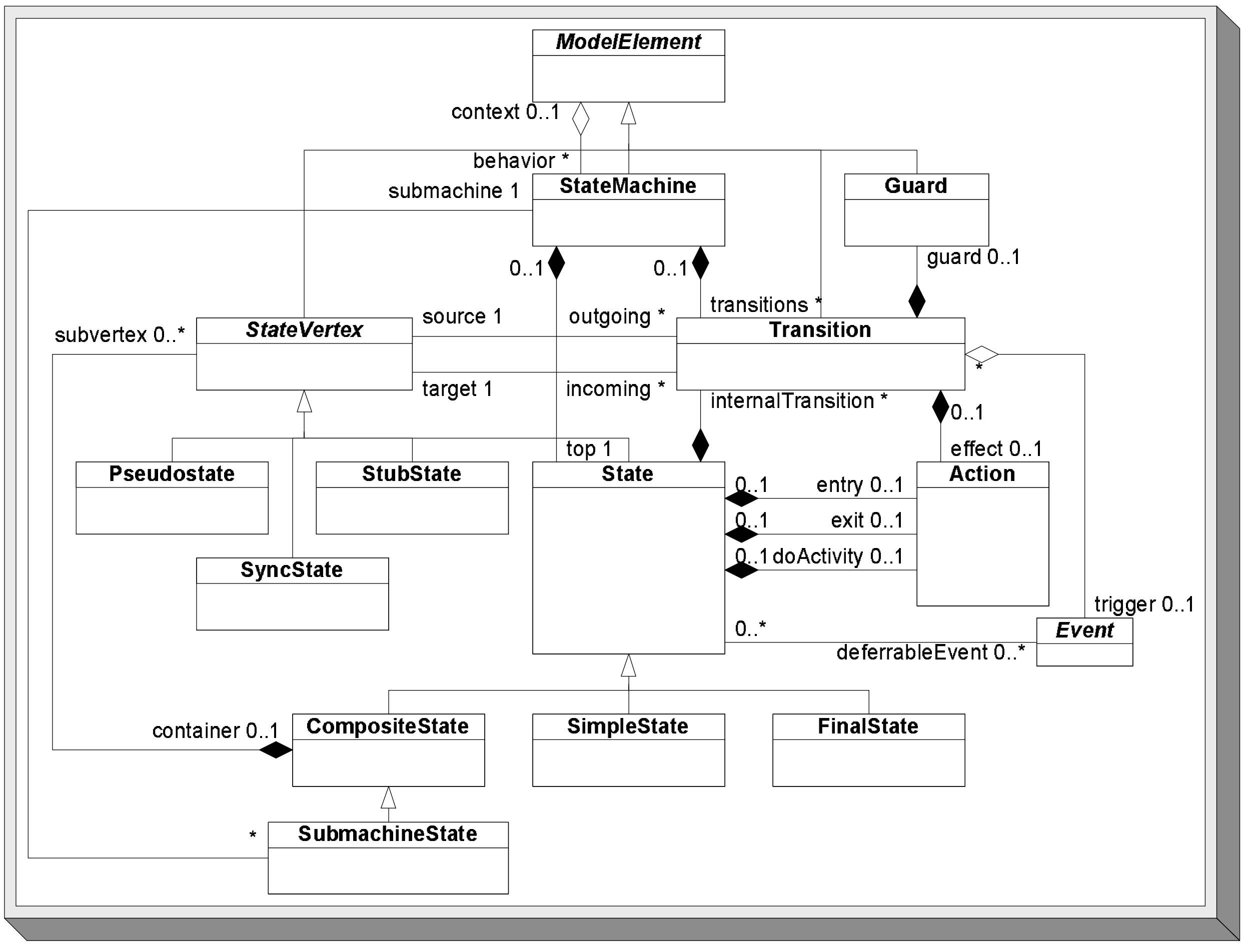


Figure 2: Metamodel of UML statecharts [10]

on the run to completion assumption i.e. an event can only be dequeued and dispatched if the processing of the previous one is fully completed.

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.

In the case of the example in Fig. 3 if the state machine is in state *s3* after the arrival of event x and the guard w evaluates to true the exit action of *s3* is executed than the action y associated to the transition is invoked finally the target states are entered. The entry action of the containing composite state *s4* is executed first, then the entry actions of the substates in the regions (*s7* is entered explicitly by the transition, state *s8* by default). The statechart in Fig. 3 is used as example in figures describing implementation patterns in the following sections.

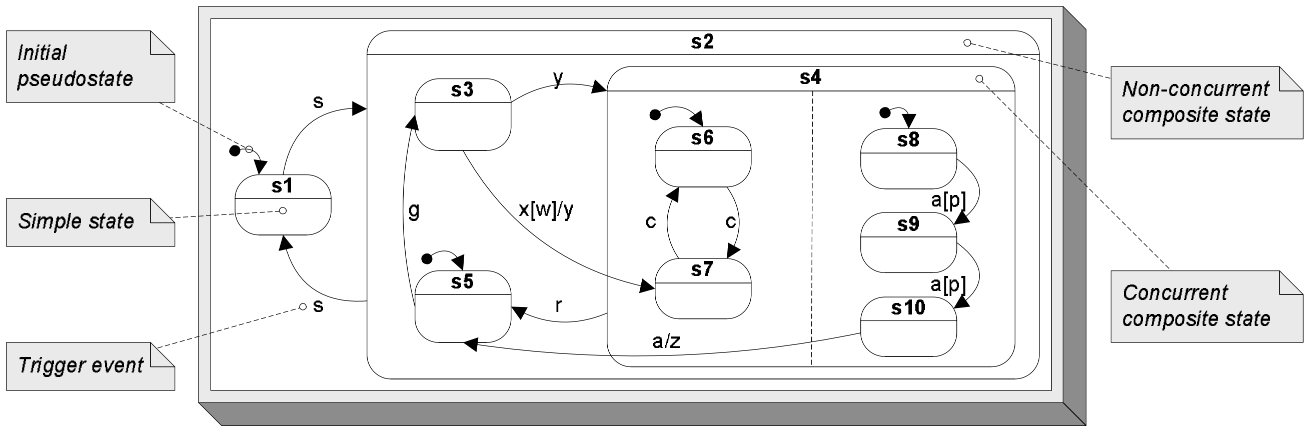


Figure 3: Example of a UML statechart diagram

In summary, the statechart package of UML is a rich toolkit that effectively supports modeling but due to its complexity it is difficult to implement. The metamodel-level representation of concurrency is not clear enough (e.g. there is no metaclass-level representation of regions) and since the transitions are allowed to cross hierarchy levels the calculation of the resulting state configuration is expensive and it is not formalized by the standard. As it will be seen in the section discussing common approaches for instantiating statecharts, the widely used techniques are unable to handle the complexity of the model. The implementation strategy developed by us is based on the transformation of statecharts to an intermediate representation reducing this way the complexity of the model, separating the concepts and providing a clearly structured model representation.

3 Common implementations of UML statecharts

This section introduces the common approaches for implementing event-driven control flow based on statechart specification. The discussion concludes that none of the well-known techniques provide complete solution for implementing concurrent operation.

3.1 Nested switch statements

The most common implementation approach uses two *nested switch statements* for partitioning the event handler function to segments reflecting the behavior of the object in specific states (external branches) and sub-segments for each event handled in the state (internal branches) (Fig. 4). These latter sub-segments are responsible for performing the action specified by the statechart in specific state – event combinations.

The most important drawback of this relatively simple and widely used structure is that it does not provide explicit means for reflecting the transition structure, state hierarchy[1] and entry/exit actions associated to states. The action chain to be performed on state transitions must be coded in the appropriate sub-segments resulting in code repetition (e.g. entry action of a state must be included in all event-handler sub-segments that may perform transition to this state).

The code organized in this way is hard to maintain (e.g. modifying a single statement in the entry action of a composite state requires the modification of all sub-segments that perform

[1]“No support” means that the design pattern was not designed to explicitly reflect a specific feature but does not mean that it can not be implemented at all.

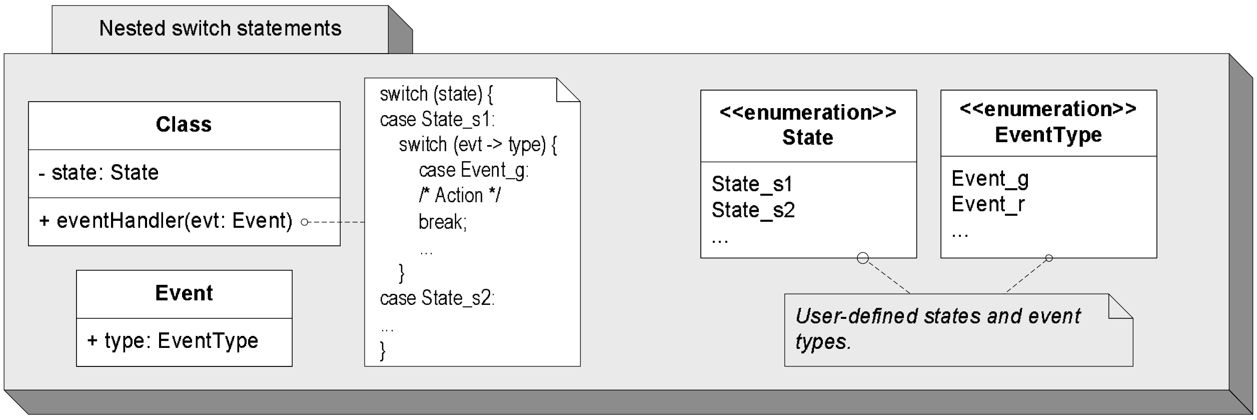


Figure 4: Implementation by nested switch statements

transition to this state or one of its substates).

The repetition of source code can be avoided by grouping the actions (entry, exit etc.) in functions, but changing the target or source state of a transition still requires modification at many places of the implementation.

Implementing and maintaining the code generated by following this pattern is error-prone and labor intensive, but efficiently usable in *automatic code generators* where the code maintenance is substituted by forward-engineering (the source code-level implementation is re-generated after modifying the high-level model). I-Logix Rhapsody [2] follows an approach similar to this pattern (with major enhancements).

3.2 Action-state tables

Action-state tables store pointers to functions to be executed on the arrival of specific events in specific states (Fig. 5). These functions are similar to the code sub-segments mentioned in the discussion of nested switch statements.

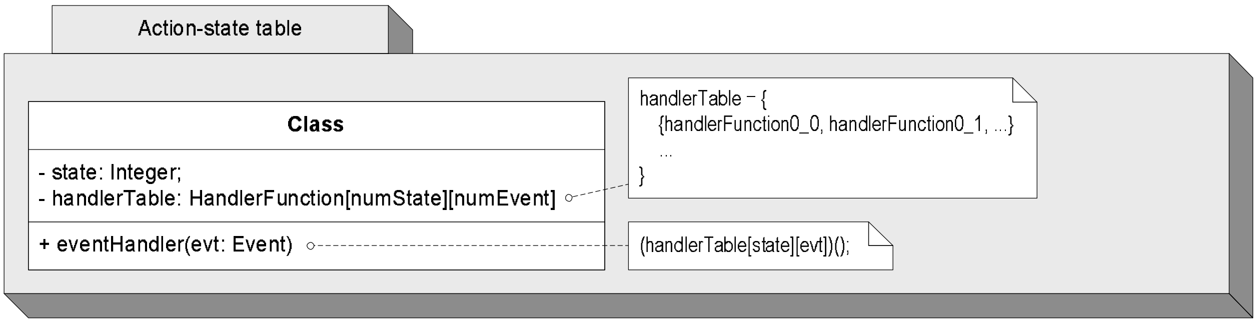

Action-state table
Class
- state: Integer;
- handlerTable: HandlerFunction[numState][numEvent]
+ eventHandler(evt: Event)
handlerTable = {
{handlerFunction0_0, handlerFunction0_1, ...}
...
}
(handlerTable[state][evt])();

Figure 5: Implementation by action-state tables

Since typically only a small minority of the events received by the object are handled in a specific state, most of the large pointer table is unused (unhandled events can be simply skipped in case of nested switch statements).

This approach is slightly faster than the nested switch structure (only one indirect call) but requires much more memory (one pointer for all state – event combinations). The other important disadvantages are the same as above: no support for hierarchy, history, concurrency, etc. and the resulting code is inflexible.

3.3 The "State" design pattern

In the object-oriented *"State" design pattern [1]* states are represented as descendants of a common interface class that declares handler functions for all events possibly received by the class (Fig. 6). The actual state is reflected by a pointer to be updated on state transitions.

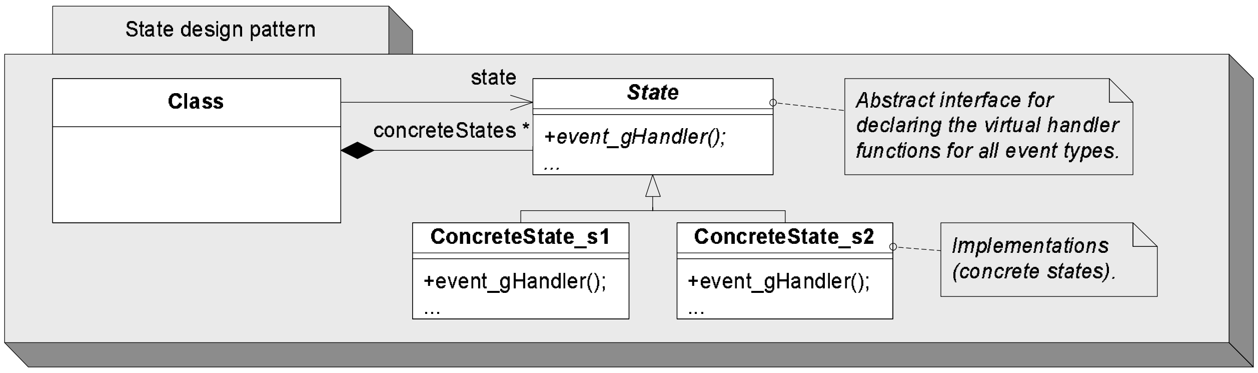
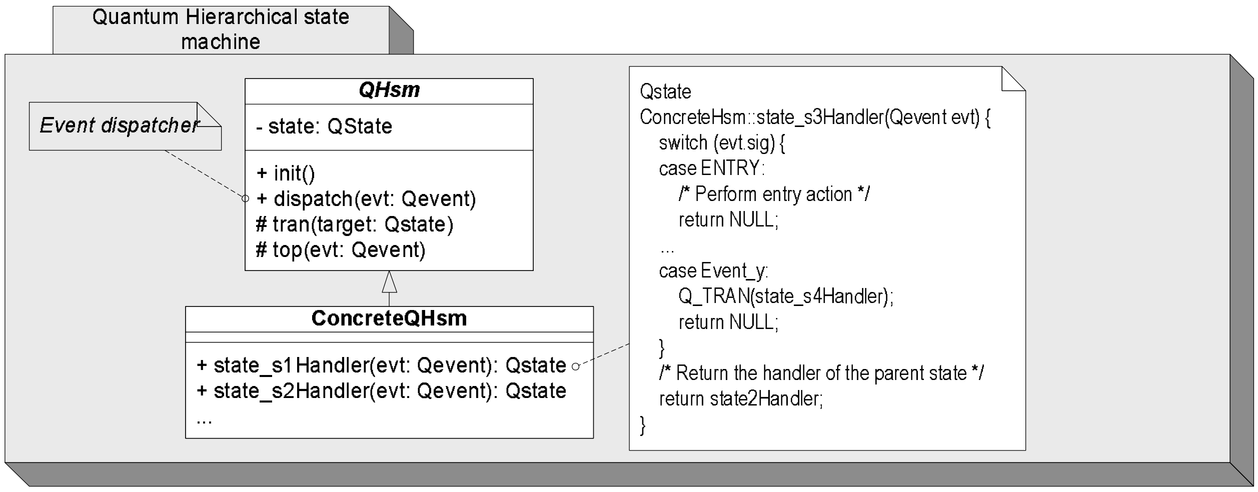


Figure 6: Implementation by the State design pattern

This pattern groups behavior associated to the specific states of the object into different classes enabling this way the separation of concerns in an elegant and efficient way. The most important weakness of this approach is that it does not provide any means for implementing the dynamic parts of the model. The action chain to be performed on state transitions as defined by the UML behavioral model must be coded in the event handler functions as seen in the previous approaches. The other deficiencies remain unsolved as well: no explicit support for hierarchy, history and concurrency.

3.4 The Quantum Hierarchical state machine

The *Quantum Hierarchical state machine* (QHsm [7]) is an improved version of the *State oriented Programming* (SoP [8]) pattern. It provides elegant solutions for representing state hierarchy and efficient and flexible implementation of transition dynamics based on handshaking of event handler functions.



```
Qstate
ConcreteHsm::state_s3Handler(Qevent evt) {
  switch (evt.sig) {
  case ENTRY:
    /* Perform entry action */
    return NULL;
  ...
  case Event_y:
    Q_TRAN(state_s4Handler);
    return NULL;
  }
  /* Return the handler of the parent state */
  return state2Handler;
}
```

Figure 7: Implementation by Quantum Hierarchical state machine

The basis of the pattern (Fig. 7) is the abstract `QHsm` class that provides implementation for the event dispatcher function and the function implementing the state transitions. Classes derived from it have to implement functions for handling events in specific states (one function for each composite and simple state). These functions return either a `NULL` pointer if the event was successfully handled in the actual state or the address of the parent state (i.e. a pointer to the parent state event handler). The dispatcher function inherited from `QHsm` is responsible for delegating events from the deepest state in the hierarchy until it is handled or the top state is reached.

State transitions can be performed by simply calling the `tran` function inherited from `QHsm` in the event handler function. It will collect the states to be exited and the ones to be entered when performing the transition, only the target state must be given as parameter. This is a very flexible implementation since modifying the target of a transition requires the modification of only one statement in the event handler of the source state – the maintenance is affordable even by hand.

Performing entry and exit actions on transitions and discovery of hierarchy is solved by sending special events to the handler functions by the `tran` method.

Although this pattern provides support for reflecting state hierarchy and effective and flexible implementation of transitions, it does not follow strictly the UML behavioral model: the action associated to the transition cannot be directly represented, it has to be performed whether before or after the entry – exit action chain. Deep history pseudostates can be implemented by using a feature of the pattern but shallow history can not. Concurrency is not supported but a strategy is suggested in [7] that can reflect some aspects of concurrent operation.

3.5 The Extended Quantum Hierarchical state machine

This subsection suggests some modifications and additional design patterns extending the Quantum Hierarchical state machine. The extensions result in an implementation strategy that supports the original UML model on state transitions (actions can be associated to transitions), shallow and deep history pseudostates and most cases of concurrent operation. Our approach is called *Extended Quantum Hierarchical state machine* pattern (EQHsm) (Fig. 8).

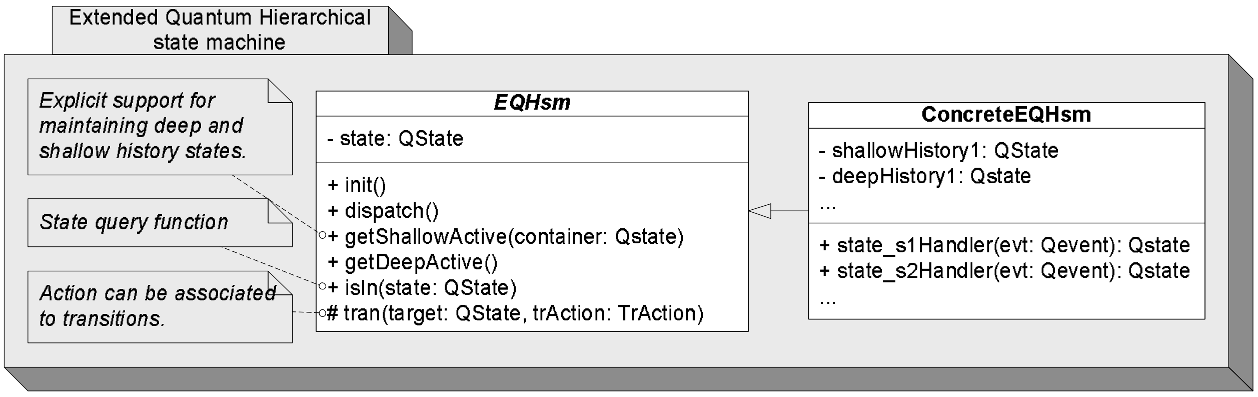

Extended Quantum Hierarchical state machine
Explicit support for maintaining deep and shallow history states.
State query function
Action can be associated to transitions.
EQHsm
- state: QState
+ init()
+ dispatch()
+ getShallowActive(container: Qstate)
+ getDeepActive()
+ isIn(state: QState)
# tran(target: QState, trAction: TrAction)
ConcreteEQHsm
- shallowHistory1: QState
- deepHistory1: Qstate
...
+ state_s1Handler(evt: Qevent): Qstate
+ state_s2Handler(evt: Qevent): Qstate
...

Figure 8: Implementation by the Extended Quantum Hierarchical state machine

Precise ordering of actions to be performed on state transitions can be achieved by a minor modification of the transition function, namely adding a new `trAction` parameter to the function signature (pointer to the function representing the action associated to the transition) and the insertion of a call to this function between performing the exit and the entry action chain.

History states can be represented as pointers to event handler functions. The `EQHsm` class provides functions for updating these pointers in the exit action of the encapsulating composite state.

Cases of concurrent operation where the set of events handled by states in a concurrent regions and the set of events handled by the containing states are disjoint (therefore no transition conflict occurs) and there are no transitions crossing the border of a concurrent region can be implemented by multiple communicating state machines with wrapper states and special events. In Fig. 9 events e_4 and e_5 belong to regions of state s_2. In case of this example the original statechart can be simulated with three automata, one for storing the top-level states (s_1 and s_2) and two automata representing one region each. Transitions targeting states in regions are substituted by transitions targeting the containing composite state with an associated action that sends an event (e.g. `_entryFork`) to the automaton representing the region to force it to the appropriate state.

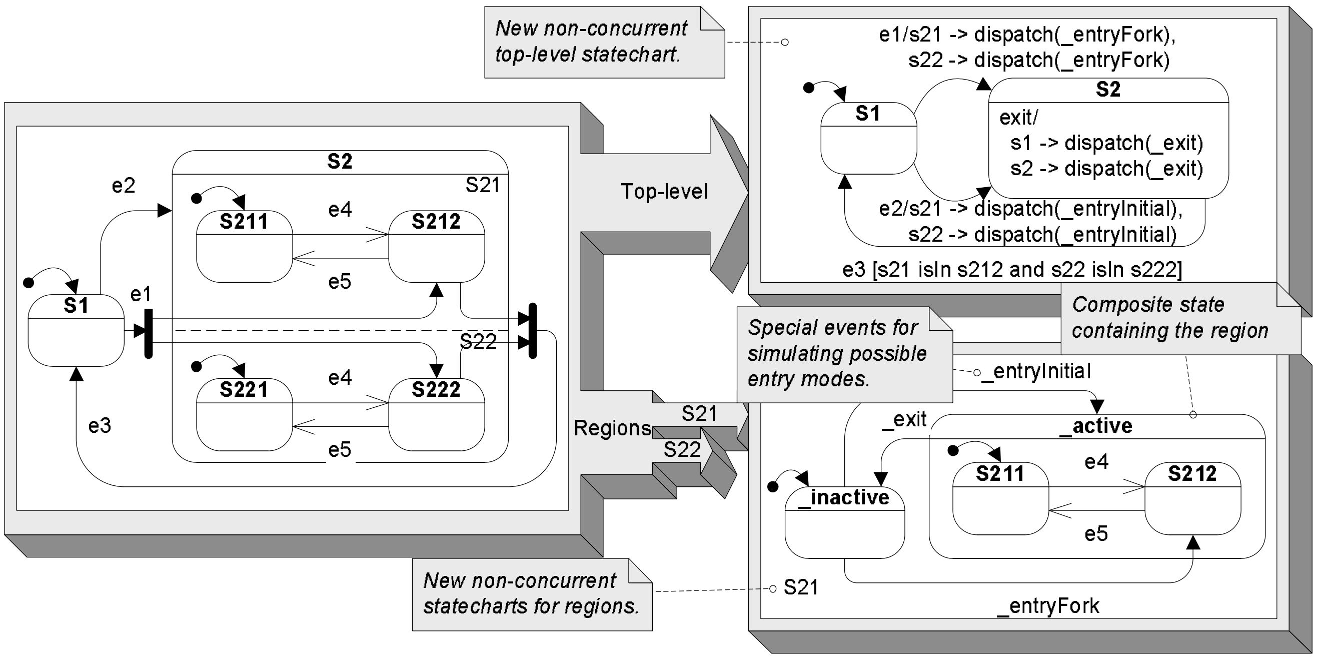

New non-concurrent top-level statechart.
e1/s21 -> dispatch(_entryFork), s22 -> dispatch(_entryFork)
S1
S2
exit/ s1 -> dispatch(_exit) s2 -> dispatch(_exit)
e2/s21 -> dispatch(_entryInitial), s22 -> dispatch(_entryInitial)
e3 [s21 isIn s212 and s22 isIn s222]
Composite state containing the region
Special events for simulating possible entry modes.
_entryInitial
_exit
_active
_inactive
S211
S212
e4
e5
S21
_entryFork
New non-concurrent statecharts for regions.
Top-level
Regions
S21
S22
S2
S21
S22
e2
e1
e3
S211
S212
S221
S222
e4
e5

Figure 9: Decomposition of concurrent composite states into individual state machines

This solution provides support for the majority of UML statechart features while posing only modest CPU and memory requirements, therefore usable in embedded systems equipped with small memory and weak CPU.

3.6 Feasibility of common approaches

The common approaches discussed here can be used for implementing only more or less restricted subsets of UML statechart features. Although not even the most sophisticated strategy is capable of correctly instantiating concurrent behavior, these strategies can be used for implementing many cases of statechart models. The Quantum Hierarchical state machine pattern and its extended variant are especially suited for implementing statecharts where automatic code generation is infeasible and thus the manual maintenance of the code is required.

The complete solution for implementing concurrent operation (transition conflict resolution, interlevel transitions to regions etc.) has remained an open issue.

4 Implementation based on the Extended Hierarchical Automaton representation

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.

4.1 Syntax

The *syntax* of extended hierarchical automata is described in a functional notation in [5], a metamodel is presented in [12] (Fig. 10). In the following a short informal overview is given mainly concentrating on the representation of UML concepts. For explained definition refer to [5] and [4].

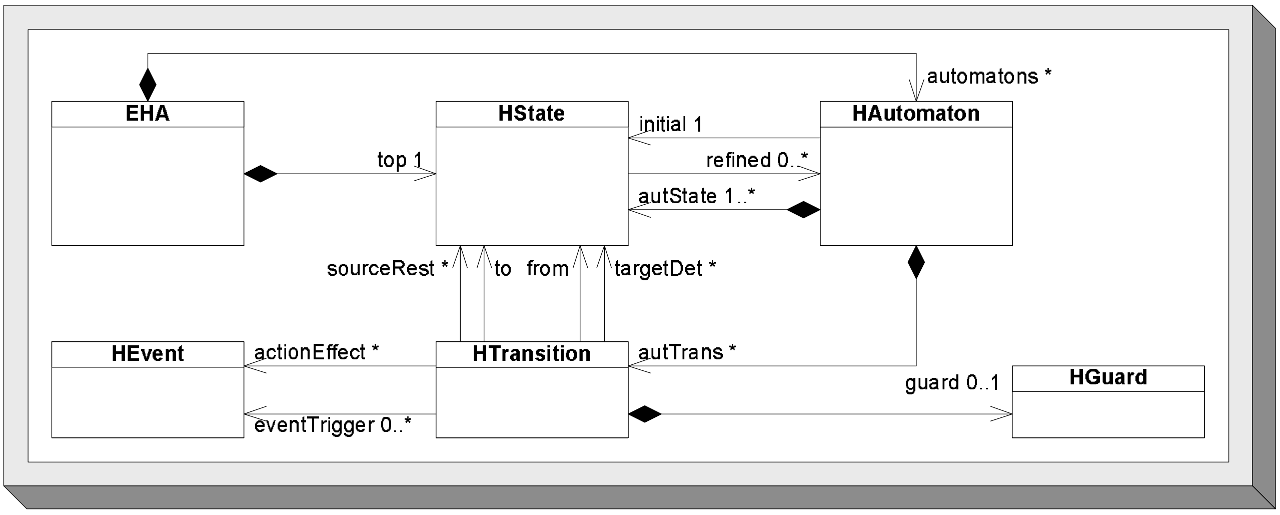


Figure 10: Metamodel of Extended Hierarchical Automata

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 replaced 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.

4.2 Operational semantics

The *operational semantics* is expressed by a Kripke-structure in [5].

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 source state and active substates in its refinement automata are exited and the target state and all states in the target determination set are entered.

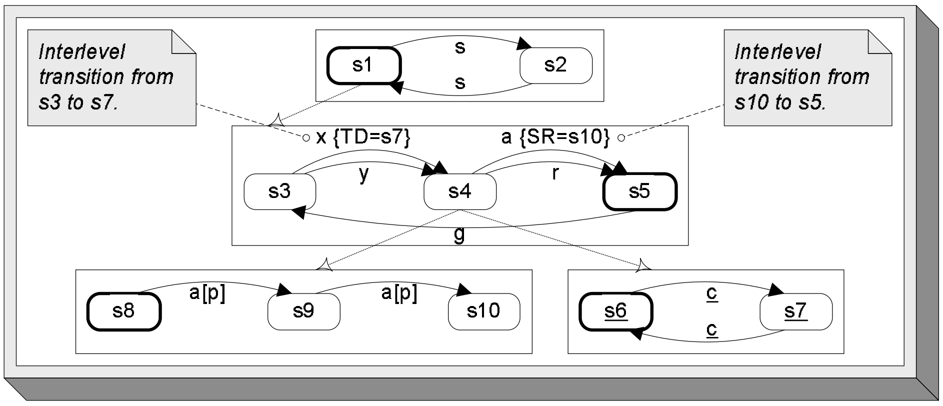

Interlevel transition from s3 to s7.
s
s1
s2
s
Interlevel transition from s10 to s5.
x {TD=s7}
a {SR=s10}
y
r
s3
s4
s5
g
a[p]
a[p]
s8
s9
s10
s6
c
c
s7

Figure 11: EHA representation of the example UML statechart

The extended hierarchical automaton of the example presented in Sect. 2 represents the statechart in a clear refinement hierarchy (Fig. 11). Statechart states (simple and composite ones) are mapped to EHA states (*s1*. . . *s10*). Concurrent and non-concurrent refinement is expressed by automata assigned to states. This way the non-concurrent state *s2* is refined to a single automaton containing the states *s3*, *s4* and *s5* while the concurrent composite state *s4* 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.

The original EHA model used by [5] and [4] does not deal with entry and exit actions since these features do not belong to the mathematical abstraction. If entry and exit actions do not generate new events their introduction does not modify the mathematical model.

4.3 The implementation pattern

The implementation 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.

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* (Fig. 12) is a modified and preprocessed form of the EHA metamodel. Modifications were taken to enable faster navigation and smaller memory consumption.

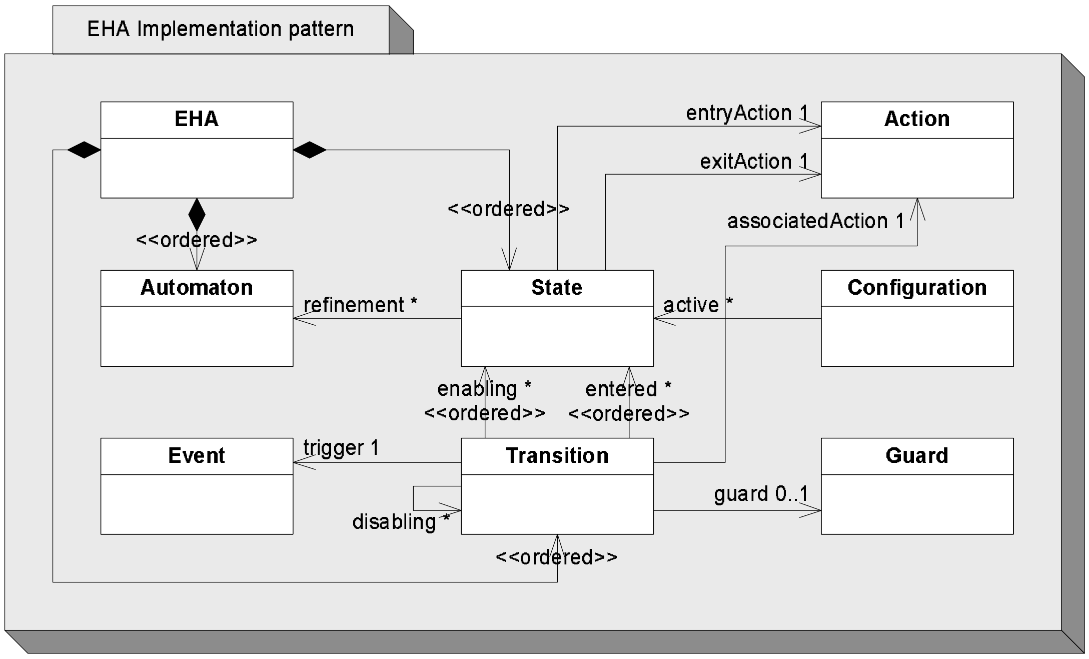


Figure 12: Class diagram of the proposed pattern

The topmost container of the static information is the `EHA` 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 containment 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.

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 `EHA` 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 instance of the class described by the statechart, the static representation is read-only therefore it can be stored in a single instance.

4.4 Dynamic behavior

The *interpreter function* is based on the formal semantics (PROMELA code) as described in [4]. 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 event 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 transitions).

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 applying 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, this way 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 of entry actions to be called would be very time-consuming. The implementation pattern proposes 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 in the list.

States to *exit* from cannot be calculated during the code generation since it depends on the actual state configuration of the object when receiving the event triggering the transition. As [10] 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.

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

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

    s.exitAction();                   // Exit action of the state
    markInactive(s);                  // Mark the state inactive
```

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

```
step(EHA eha, Configuration cfg, Event e)
    enabledSet = collectEnabled();     // Collect enabled transitions
    fireableSet = collectFireable();   // Collect fireable transitions

    forEach t in fireableSet           // t is a fireable transition

        recursiveExit(t.source);       // Recursive exit from the source

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

        forEach s in t.entered         // s is a state to be entered
            s.entryAction();           // Entry action of s
            markActive(s, cfg);        // Mark the state active
```

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

5 Code generation framework

The Extended Hierarchical Automaton equivalent of a statechart can be used as common intermediate representation used for both code generation and model checking enabling this way the automatic implementation of formally verified models (Fig. 13).

A transformation method from EHA to the Process Meta Language (PROMELA, the specification language of the SPIN [3] model checker) is proposed in [4]. Automatic code generation can be based on the implementation pattern for Extended Hierarchical Automata described in this paper.

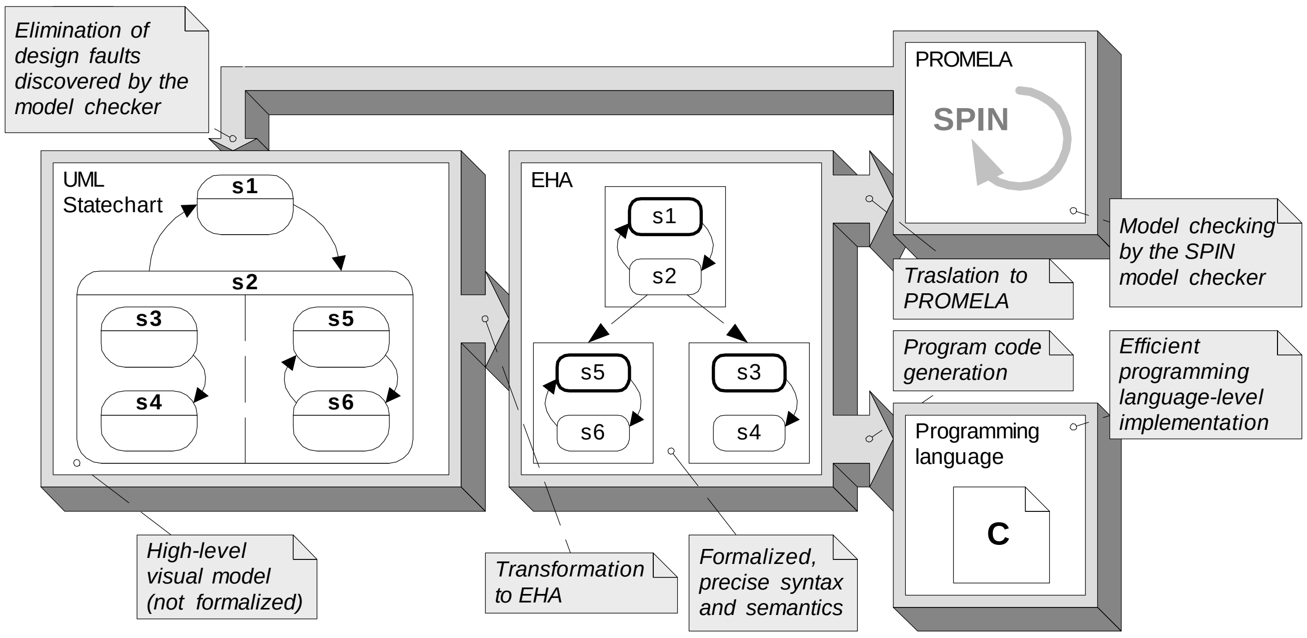

Elimination of design faults discovered by the model checker
PROMELA
SPIN
Model checking by the SPIN model checker
UML Statechart
s1
s2
s3
s4
s5
s6
EHA
s1
s2
s5
s6
s3
s4
Traslation to PROMELA
Program code generation
Efficient programming language-level implementation
Programming language
C
High-level visual model (not formalized)
Transformation to EHA
Formalized, precise syntax and semantics

Figure 13: Model checking and automatic code generation unified in a single framework

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. According to our calculations in the case of the example presented in this paper 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 10 states in the model the configuration information of an object fits in 10 bits.

6 Testability considerations

Efficient testing and self checking requires additional methods that do not directly belong to the implementation of the control core:

- Query functions for reliably *reporting* about the actual *state configuration* of the application. This requirement can be satisfied by providing functions for interpreting the bit vector describing the dynamic configuration.

- State invariant checks to enable the *detection of inconsistency* between the explicit state (value of the bit vector) and the actual value of the object attributes (e.g. the high-level state of a valve controller is “Closed” while the valve is open). The code generator can be used for translating OCL [9] constraints into functions that evaluate the invariants belonging to the actual configuration and report the invariant violations.

- Temporal constraints assigned to internal data structures can be checked by automatically generated assertions.

- Concurrent control flow checking can be based on an embedded statechart-level watchdog [6]. Watchdogs are relatively simple coprocessor-like applications that are used to check the control flow of the main processor or the tested application. In our case a software-implemented watchdog can be applied to detect the deviations of the control flow traversed by the tested program from the legal sequences of transitions specified by the statechart. This approach necessitates the insertion of signature transfer operations in the entry actions of the states. This can be implemented automatically by the code generator.

- More sophisticated constraints can be assigned to the software control flow by using temporal logic extensions of the OCL [11] (e.g. possible accepting and resulting states on the arrival of specific events, avoidance of hazardous configurations etc.). These requirements can be met by extending the capabilities of the embedded watchdog by adding memory to it enabling the observation of multi-transition sequences.

7 Conclusion and future work

This article outlined our investigation aiming at discovery of an efficient and sophisticated implementation methodology for UML statecharts. An implementation pattern was developed that is capable of implementing a major subset of UML features therefore providing a stable and efficient control backbone for systems instantiated from it. It provides methods for self checking and testing support as well. Our next step on this way will be the development of a hierarchical control flow checking and error reporting method.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[2] I-Logix. *Rhapsody.* `http://ilogix.com`.

[3] Bell Labs. *SPIN.* `http://spinroot.com`.

[4] Diego Latella, István Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. In *Formal Aspects of Computing*, volume 11, pages 637–664. Springer Verlag, 1999.

[5] Diego Latella, István Majzik, and Mieke Massink. 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*, pages 331–347, Firenze, Italy, February 1999.

[6] István Majzik, Judit Jávorszky, András Pataricza, and Endre Selényi. Concurrent Error Detection of Program Execution Based on Statechart Specification. In *Proc. 10^{th} European Workshop on Dependable Computing*, 1999.

[7] Miro Samek. *Practical Statecharts in C/C++.* CMP Books, 2002.

[8] Miro Samek and Paul Y. Montgomery. State Oriented Programming. *Embedded Systems Programming*, 2000.

[9] OMG. *Object Constraint Language Specification.* 2001.

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

[11] Stephan Flake and Wolfgang Mueller. An OCL Extension for Real-Time Constraints. *Lecture Notes in Computer Science*, 2002.

[12] Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, August 2002.