Impact of Statechart Implementation Techniques on the Effectiveness of Fault Detection Mechanisms

G. Pintér and I. Majzik
Department of Measurement and Information Systems
Budapest University of Technology and Economics
{pinter, majzik}@mit.bme.hu

Abstract
This paper presents the analysis of an experiment series aiming at the discovery of the impact of two inherently different statechart implementation methods on the behavior of the resulting executables in the presence of faults. The discussion identifies the key features of implementation techniques influencing the effectiveness of standard fault detection mechanisms (memory protection, assertions etc.) and an advanced statechart-level watchdog scheme used for detecting the deviations from the abstract implementation-independent behavioral specification.

1. Introduction
When selecting components for dependable systems the dominant requirements are not only the low price and high performance but the dependability of the resulting application as well. For example in case of software for railway control designed according to EN-50128, if COTS software is to be used in higher software integrity levels, then the following precautions shall be taken: (1) an analysis of possible failures shall be carried out and (2) a strategy shall be defined to detect failures of the COTS software and to protect the system from these failures. Fault detection mechanisms are the substantial facilities for developing fault tolerant systems. As automatic code generation plays a more and more central role in modern software development methodologies, the impact of code generation patterns on the effectiveness of fault detection mechanisms is becoming an important issue. This paper presents an experiment series aiming at investigation of two different statechart implementation methods and identifies the key differences between behaviors of applications in the presence of faults.

Several methods were proposed in the literature for instantiating UML statecharts in programming languages. These techniques were reported to significantly differ in CPU requirements, memory consumption, maintenance costs etc. Two inherently different methods were selected that impose specific code and data section layouts. One of them implements the statechart mainly by specialized functions while the other one is based on a generic interpreter and uses constant data structures for expressing the behavioral specification.

Exhaustive fault injection experiments were performed by inverting single bits of the resulting binary executables. The modified programs were executed on a UNIX platform and the observations reported by the software and hardware fault detection mechanisms were registered. Besides the standard mechanisms provided by the CPU, the memory management unit and common software assertions, an advanced form of software fault detection mechanisms, a statechart-level watchdog processor [15] was applied. The watchdog receives signatures indicating behavior at the abstraction level of the statechart (e.g. selecting a transition for firing, entering a state etc.) and verifies the acceptability according to the statechart. The signature transfer is implemented by instrumenting the applications.

The key sections of the article are as follows: Section 2 outlines the implementation patterns, Section 3 presents the environment used for investigating the behavior of application in the presence of faults, Sections 4 and 5 introduce the observations obtained from the experiment series, finally Section 6 summarizes the results.

2. Preparing benchmark applications
This section presents the benchmark application used in our experiments, introduces the implementation patterns used for instantiating UML statecharts in C/C++ and highlights the differences between the code and data characteristics of the resulting executables. The goal of these introductions is to enable the analysis of measurement results
with respect to the binary layout of the executables, therefore most of the implementation details are not considered here.

2.1. The benchmark application

In our experiments the implementation patterns were applied to the non-trivial statechart of the desktop calculator example (Fig. 1) presented in [16]. The statechart consists of a state hierarchy of 16 states (and the invisible top state), 30 transitions and is driven by 10 event classes (guards and actions were suppressed in the figure for simplicity). The events represent buttons of the calculator: numbers (num19), operators (oper), decimal point (point), equals sign (equals), the percent symbol (percent), the clear (c) and the cancel entry (ce) buttons.

The topmost operational state is calc. The self transition triggered by the clear button (c) takes the calculator back to its initial state configuration while the reception of the exit event switches the calculator off. The ready state is refined into two substates, result (displaying the result of the previous computation) and begin (the calculator was just switched on or was taken back to the initial configuration). The first (second) operand is read in state operand1 (operand2) refined to substates representing the partially typed operands (zero, integer and the fraction parts). The operator buttons bring the calculator to state opEntered (state between reading the two arguments). The operands may be negative (preceded by a minus sign) as represented by the negated states.

2.2. The Quantum Hierarchical state machine pattern

The Quantum Hierarchical state machine (QHsm) pattern is a central concept of the Quantum Framework [16], a complete implementation environment designed to work in real-time embedded applications. The key artifacts are the QHsm base class (from the framework) and the classes derived from it by the programmer (Fig. 2). The QHsm class implements the most important methods for event dispatching (dispatch) and state transitions (tran).

States are represented by member functions (called state functions in this discussion) in derived classes that receive the event to be processed as argument and return NULL if the event was successfully handled or the pointer to the member function representing its parent state. In case of the example in the figure the state function zero1 in class Calculator returns the address of operand1 if it does not handle an event (e.g. the pressing of an operator button) since the
parent state of zero1 is operand1. This handshaking enables
the propagation of events to higher abstraction (state hierar-
chy) level until they are handled or disposed at the top level.
The dispatcher function (dispatch) inherited from QHsm is
responsible for delegating events upwards from the deep-
est state in the hierarchy (i.e. calling the function pointers
until NULL is returned). The type of pointers to state func-
tions is defined as QState. The QHsm class maintains a var-
iable myState of this type for storing the actual state. State
functions typically contain a switch statement for branch-
ing according to the event to be processed.

Transitions are performed in state functions by inserting
the preprocessor macro Q_TRAN that calls the tran method
inherited from QHsm. Tran discovers the state hierarchy by
sending special events to state functions, collects the states
to be exited and the ones to be entered when performing
the transition. Only the target state must be given as param-
eter. The discovered hierarchy is cached in the static vari-
ables inserted by the macro.

The C++ version of the original QHsm implementation
was ported to UNIX and instrumented in order to enable
the run-time investigation by a watchdog [15]. The modified
transition and event dispatcher functions inform the watch-
dog about the start and finish of event processing and about
states that are exited or entered during a transition.

2.3. The EHA2C pattern

The pattern proposed in [14] is based on transform-
ing UML statecharts to Extended Hierarchical Automata
(EHA) [5] [6] and applying a scheme for mapping the be-
havioral specification to the ANSI C language. The ap-
proach is inherently different from the QHsm pattern:
EHA2C is based on a single interpreter function and a com-
plex constant data structure representing the EHA. This
subsection briefly introduces Extended Hierarchical Au-
tomat focusing on the differences from the UML state-
chart model and outlines the code generation method.

An EHA consists of sequential automata. A sequen-
tial automaton contains simple (non-composite) states and
transitions between them. EHA states represent simple and
composite states of the UML model and can be refined to
any number of sequential automata. All the refining au-
tomata are running concurrently (i.e. concurrent com-
posite states are modeled by EHA states refined to several
automata representing one region each).

Source and target states of an EHA transition are al-
ways in the same automaton. UML transitions connecting
states at different hierarchy levels are represented by tran-
sitions labeled with special guard sets containing the origi-
nal source and target states called source restriction and tar-
get determination respectively.

The EHA concepts can be represented by the data struc-
ture described in Fig. 3. The EHA2C pattern uses a compact
C representation of this construct for storing EHA models.

The topmost container of the static information is the
EHA class. The containment relation between automata and
states, the state refinement, transition triggers and guards,
entry and exit actions etc. are represented by associations
and aggregations of the class diagram in the obvious way.
The ordered stereotype of associations targeting automata,
states and transitions indicate that these aggregations should
be implemented by arrays since the positions of the ob-
jects (i.e. the array index) are used as unique identifiers
amongst objects of the same class (transitions, states etc.).
The source (target) state and source (target) determination
sets are unified into the enabling (entered) sets. The priority
relation between transitions is statically calculated and
stored in a self-association with the role disabling.

Having outlined the high-level structure the next key as-
pect is the mapping of abstract concepts (classes, associa-
tion etc.) to the C programming language. Guards and ac-
tions are implemented by functions therefore the associ-
actions targeting them are implemented by function pointers. Events are represented by an enumerated type. Associations targeting automata, states and transitions are implemented by identifier arrays (i.e., not by pointers since identifiers are generally shorter than pointers). The identifier arrays are collected into compound arrays reducing this way the storage space loss introduced by the word-alignment of data structures. The state configuration is represented by a bit vector (the single non-constant data) where the $i^{th}$ bit is true exactly if the state with identifier $i$ is active.

The interpreter function is independent of the actual EHA model. It takes the static data structure, the actual configuration and the event to be processed as argument, calculates the step to be taken, calls the appropriate functions representing state entry and exit actions or actions associated to transitions and updates the configuration. In this pattern the interpreter is the only function to be instrumented.

## 2.4. Binary layout

The implementation patterns have an important impact on the binary image of the resulting executables (size and content of code and data sections etc.) as shown in Fig. 4.

The QHsm pattern is based on a relatively complex base class and the descendant classes containing one function for each state resulting in a relatively large code segment (i.e., larger than the single interpreter function of the EHA2C pattern not taking into consideration the implementations of the actions). These state functions consist of a switch statement that performs an indirect jump with respect to the actually processed event to the code fragment representing the appropriate case edge. Jump target addresses are typically stored in an initialized array in the code section by the C compilers.

The data part is very compact in case of the QHsm pattern: the QHsm class maintains some pointers representing the actual state and the static pointer arrays are used for caching the transitions as outlined above. Pointers are used directly for accessing functions.

## 3. Investigation in the presence of faults

While the previous section illustrated the differences between the two implementation patterns, this section describes the method we were using for investigating the impact of these differences on the behavior of applications in the presence of faults and the hardware and software fault detection mechanisms that were available during the experiments.

### 3.1. Fault injection experiments

Fault injection corresponds to the artificial insertion of faults into computer systems [17] [4] aiming at (1) understanding the effects of faults in the system, (2) enhancement
of fault tolerance methods and (3) measurement of coverage provided by the fault tolerance mechanisms [3].

Fault injection experiments were successfully applied for dependability validation of hardware designs and software architectures [9] [2] [1]. Since the role of implementation patterns is determining the size and content of code and data sections, our investigations aimed at discovery of behavior of applications in the presence of impairments of these regions.

The sources according to the two patterns were built on a UNIX operating system running on a PC platform resulting in binaries of 53 Kbytes (EHA2C) and 73 Kbytes (QHsm). Exhaustive software-implemented fault injection (SWIFI) experiments [8] were performed on both binaries by injecting single bit inversions in all bit positions (one in each experiment runs). The modified binaries were executed and the outputs of the fault detection mechanisms were collected into a database.

3.2. Hardware fault detection mechanisms

The CPU and the memory management unit (MMU) provide several means for detecting the faults in the execution of tasks in a modern computer system. Most of the exceptions indicated by the hardware are mapped to UNIX signals. If the CPU (FPU) fetches an invalid instruction, the operating system handles the exception and sends an Illegal instruction (Floating-point exception) signal to the application. Possible sources of these signals are the impairment (e.g. bit inversion) of an instruction (operation code) or an attempt for executing an area that does not contain instruction placed there by the compiler possibly caused by a bit inversion in an argument (i.e. target address of the previous branch instruction).

UNIX processes live in protected virtual memory segments. An attempt for accessing an area outside the allocated region results in a MMU exception mapped to the “Segmentation fault” signal. Since memory corruption faults originate mainly from incorrect pointer handling and branch instructions targeting non-code area, the MMU signals indicate mainly corruption of data structures (e.g. faulty pointer operations).

3.3. Software fault detection mechanisms

Algorithms implemented in software typically can not be applied to all inputs. The event dispatchers and interpreters used in the implementation patterns contain several assertions aiming at detection of these cases and signaling a failure (e.g. the EHA2C interpreter performs a basic check to ensure that the identifiers lie within a valid interval before using them as array indices). Since pointers are holding addresses of functions or dynamically allocated memory objects, asserting pointers is not as straightforward as checking, whether programmer-assigned identifiers lay within a valid interval or not.

Detecting the deviation from the abstract behavioral specification is a non-trivial issue. There were several watchdog schemes [10] proposed in the literature aiming at the concurrent supervision of program execution based on comparing the actual execution of the application to a reference control flow graph extracted from the assembly language-level implementation of the application. The control flow specification can be stored in the watchdog [12], [7] or embedded in the signatures sent by the application [13] [11].

Although these implementations were successfully applied for detecting transient hardware faults resulting in deviations from the correct flow of machine instructions within functions, method calls and interrupt invocations, their state model can not express hierarchies and concurrent execution.

![Diagram](image)

**Figure 5. Supervision of statechart implementations**

In our case a statechart-level watchdog application [15] was developed that receives messages from the application indicating the construction and destruction of objects, entering and exiting states, start and finish of the initialization phase and the event processing etc. The watchdog is capable of detecting several fault classes including attempts for selecting transitions that are not triggered by the actually processed event, entering (exiting) already active (inactive) states, performing disabled transitions, violation of the order in which states are to be entered (exited) in case of entering (exiting) composite states, entering (exiting) states that
The effectiveness of fault detection mechanisms heavily depends on the fault model of the environment and the implementation characteristics of the application. Hardware measures discussed here are effective in detecting low-level impairments (e.g. transient hardware faults, memory corruptions etc.) resulting in attempts for executing an invalid instruction or accessing a memory region outside the address space of the process. Assertions and software sanity checks are usable for detecting cases were an algorithm can not be applied to a faulty input data set or serious data corruptions resulting in invalid identifiers or pointers.

The statechart level watchdog represents the highest abstraction level of software fault detection mechanisms. It aims at detecting violations of the statechart semantics possibly resulting from data or code section impairments that did not trigger errors detected by mechanisms operating on lower abstraction levels.

Note that while the hardware mechanisms are not always available on less sophisticated platforms (e.g. low-end microprocessors and microcontrollers do not implement a protected virtual memory scheme) the software fault detection mechanisms are still applicable.

4. Fault injections into the code sections

The code sections of the benchmark executables contain several functions that are not related to the implementation patterns (e.g. initialization, communication with the watchdog). The relevant regions belonging to the implementation patterns were identified by disassembling the executables and calculating the corresponding file offsets (relevant intervals in the executables).

Faults can only be detected if they become effective (i.e. the CPU processes the faulty byte resulting in an erroneous state of a system component). Focusing on experiment runs where the injected faults were effective is a non-trivial issue that could be addressed by building a map of memory regions that were fetched as instructions (and their arguments) or read as data into a register. In our experiments a custom debug utility was used for executing the benchmark applications instruction-by-instruction and collecting the bytes that were fetched as instructions or arguments. Although this approach still lacks the capability of selecting the bytes read as data into registers, the coverage can be considered relatively high since there are typically very few bytes in the code regions used for storing constant data.

Our first observation indicated that the percentage of the effective faults was much higher in case of the EHA2C implementation (84.19%) than in case of the QHsm pattern (51.88%). The observation is explained by the fact that the QHsm base class contains some methods that were never called this way the faults injected in that region never became effective.

Figure 6 shows the relation of the non-detected effective code section faults and the ones detected by the most important mechanisms (some cases of minor relevance e.g. timeouts were suppressed). Several observations can be derived from the results:

- There were no important differences between the effectiveness of specific fault detection mechanisms with respect to the implementation patterns: 55.86% (QHsm) and 59.83% (EHA2C) covered by hardware mechanisms only (HW only in the figure) while 19.16% (QHsm) and 13.01% (EHA2C) detected by all software measures (All SW methods). Although the content of code sections is different, the fault detection

<table>
<thead>
<tr>
<th></th>
<th>Hardware</th>
<th>Watchdog</th>
<th>Assertions</th>
<th>All SW methods</th>
<th>Effective detected faults (%)</th>
<th>Detected faults (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7570</td>
<td>196</td>
<td>596</td>
<td>961</td>
<td>26.86</td>
<td>81.80</td>
</tr>
<tr>
<td>Non-detected</td>
<td>23.34</td>
<td>55.86</td>
<td>5.79</td>
<td>7.09</td>
<td>19.16</td>
<td>59.83</td>
</tr>
<tr>
<td>HW only</td>
<td>–</td>
<td>72.87</td>
<td>7.55</td>
<td>9.25</td>
<td>–</td>
<td>81.80</td>
</tr>
<tr>
<td>Watchdog only</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.66</td>
</tr>
<tr>
<td>Assertion only</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.44</td>
</tr>
<tr>
<td>All SW methods</td>
<td>25.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17.79</td>
</tr>
</tbody>
</table>

**Figure 6. Effectiveness of fault detection mechanisms in case of faults in the code sections**
mechanisms do not seem to be more or less effective with respect to the specific patterns.

- It should be highlighted that the effectiveness of the software fault detection mechanisms was remarkable (nearly 20% in case of the EHA2C pattern and 13% in case of QHsm with respect to all effective faults). Software mechanisms can play a key role in application areas where the sophisticated hardware mechanisms are not available (low-end microprocessors, embedded applications).

- According to the experiments the percentage of the non-detected faults was reduced from 29% to 23% by applying the statechart-level software watchdog in case of the EHA2C pattern and from 32% to 27% in case of the QHsm pattern respectively.

5. Fault injections into the data sections

Data segments contain not only the variables introduced by the application of the implementation patterns but some other global and static variables as well. Unfortunately the selection of faulty data related to the implementation of the state-based behavior (i.e. effective data faults) is not as straightforward as the selection of relevant regions in the code segments. The investigations presented here correspond to the entire initialized data area in the binary executables therefore the percentage of effective faults was not calculated.

Figure 7 shows the relation of the non-detected faults and the ones detected by the most important mechanisms. Our key observations based on the results are as follows:

- Important differences were identified between the effectiveness of specific fault detection mechanisms with respect to the implementation patterns. More than a factor of two in case hardware: 15.34% (EHA2C) and 39.09% (QHsm) of software mechanisms: 22.86% (EHA2C) and 7.65% (QHsm).

- In case of the QHsm pattern the hardware mechanisms provided higher fault coverage (HW: 83.39%, SW: 16.31% with respect to the detected faults). The observation can be explained by investigating the effects of faults in pointers used by the QHsm class and its descendants. The inversion of a relatively high bit of a pointer triggers a segmentation violation exception when used for addressing or branching while the calling of a “function” identified by a pointer with an inversion of a relatively low bit is likely to result in a jump to a valid but unaligned address (i.e. “between instructions”) resulting in an invalid instruction signal.

- In case of the EHA2C pattern the software mechanisms were more effective (HW: 40.08%, SW: 59.72% with respect to the detected faults). The phenomenon is explained by the content of the data section implied by the pattern: the constant identifier and index arrays are not used directly for addressing but as offsets after a validity checking. The impairment of high bits of array indices can be detected by assertions (e.g. a faulty state identifier is higher than the number of states in the entire model) before resulting in an invalid memory access. Inversions of low bits are likely to result in deviation from the specified behavior (e.g. selecting a state not entered by the transition) that can be detected by the watchdog.

- The experiment results showed that the percentage of the non-detected faults can be reduced by 8.22% when applying a software watchdog in case of the EHA2C pattern and by 3.49% in case of the QHsm pattern respectively. Note that the relative high percentage of the non-detected faults is caused by non-effective fault injections hitting initialized global or static variables that were never used (e.g. version strings etc.).
The most important conclusion of these experiments is the observation that faults in the data area of the EHA2C pattern are more likely to be detected by the software fault detection mechanisms while the impairments of the pointers used by the QHsm pattern can be addressed by the hardware mechanisms.

6. Conclusions

This article illustrated that the implementation method used for instantiating behavioral models has an important impact on the effectiveness of fault detection mechanisms that should be taken into consideration when selecting components to be built in into dependable systems.

Our experiments highlighted that the impairments of the code sections are mainly detected by hardware mechanisms (illegal instruction or segmentation violation signals) independently of the implementation method. Detection of faults in the data sections require different mechanisms in cases of the two patterns: faults of data used directly for addressing in case of the QHsm pattern can be detected by the memory management unit (hardware) while the bit inversions in the complex interpreted data structures used by the EHA2C pattern can be targeted by software mechanisms (watchdog, assertions).

The effectiveness of software fault detection mechanisms should be taken into consideration when developing software for low-end microprocessors or microcontrollers (e.g. for embedded systems) where the advanced hardware mechanisms (memory protection) are not available. The software fault detection mechanisms provided higher coverage while hardware mechanisms were found to be less effective in case of applying the EHA2C pattern. This observation does not mean that application of the EHA2C pattern explicitly requires the usage of software fault detection mechanisms since the probability of impairments of constant data structures used by the EHA2C pattern can be reduced by storing them in ROM.

Our future goals include the investigation of the effectiveness of fault detection mechanisms in case of embedded systems, application of other implementation patterns and improving the accuracy of calculations presented here by developing a mechanism for selecting data faults that become effective during the execution.

References