Modeling and Analysis of Exception Handling by Using UML Statecharts

Gergely Pintér and István Majzik

Department of Measurement and Information Systems
Budapest University of Technology and Economics, H-1521 Budapest, Hungary
{pinterg, majzik}@mit.bme.hu

Abstract. Our paper aims at proposing a framework that allows programmers to exploit the benefits of exception handling throughout the entire development chain of Java programs by modeling exception handling in the abstract UML statechart model of the application, enabling the use of automatic model checkers for checking the behavioral model for correctness even in exceptional situations, and utilizing automatic code generators for implementing the Java source of exception-aware statecharts.
Keywords: Exception handling, UML, formal methods, model checking

1 Introduction

Using software exceptions has become the de-facto way for handling abnormal situations in software, especially since the introduction of the Java language, where exceptions are part of the language, libraries and frameworks. Despite of the popularity of this mechanism development methods usually do not enable the developers to be aware of exceptional situations throughout the entire development chain (modeling, model checking and implementation).

When using the latest supported version of UML (1.5), the only way for *modeling behavior* in exceptional situations is the implicit use of the class and interaction diagrams [1] (for modeling the exception classes and their propagation respectively). The new major version of UML (2.0) introduces constructs for modeling exceptions in activity and sequence diagrams. There were several *model-checking* methods introduced for validating event-driven behavioral specifications [2]. Although these methods are capable of processing most of statechart constructs, no comprehensive approach was proposed that enable the checking of behavior in exceptional situations.

Since the *implementation* of complex event-driven behavioral logic specified by UML statecharts is labor intensive and error-prone, design patterns and code generation techniques were proposed [3, 4, 9] for automatically generating the control core just to be customized by the programmer (i.e., implementing the necessary actions, guard predicates etc.). Unfortunately these methods are not aware of exceptions: programmers can not use the normal exception handling mechanisms for indicating abnormal situations and initiating statechart-level handling mechanisms.

According to our knowledge, although there exist solutions for important problems (e.g. [5, 6, 7, 10]), no general framework has been proposed that is capable of supporting the exception handling paradigm throughout the *entire development chain* i.e., *modeling exceptions* and the handling mechanism in UML statecharts, enabling the automatic *model-checking* of the exception-aware statecharts and automatically *generating source* code where the programmer is allowed to use the standard exception throwing facilities of the language for initiating statechart-level reaction to the exceptional situation. Our paper aims at filling this gap by (1) proposing a light-weight notation for expressing the occurrence and handling of exceptions in UML statecharts, (2) discussing how to use the Extended Hierarchical Automaton notation (mathematical formalism used for model-checking statecharts) to enable the model-checking of exception-aware statecharts and (3) presenting an implementation pattern for statecharts where programmers are enabled to use exceptions for indicating exceptional situations and initiating statechart-level reaction. Although our discussion focuses on Java the implementation can be ported to other languages. The organization of our paper corresponds to the questions answered by the approach:

- *How to represent exceptions in the behavioral model and how to handle them similarly to programming language-level exception (e.g., try-catch-finally blocks, propagation etc.)?* – On one hand the *representation* of programming language-level exceptions in the statechart is achieved by catching the exceptions thrown in the programmer-written parts at the interfaces of the event dispatcher and transforming them to exception events (i.e., ordinary events in the statechart that correspond to exceptions). This approach assumes a callback-style framework organization i.e., where the programmer-written parts are called as functions by the event dispatcher. On the other hand the most important organization ideas and benefits of try-catch-finally constructs can be elevated to the *handling of exception events* by applying the statechart organization pattern suggested in this paper (Section 2).
- *How to check the exception-aware behavioral model by legacy model checkers?* – Since our approach does not modify the statechart semantics, legacy *model checkers* developed for statecharts are usable as discussed in Section 3.
- *How to implement exception-aware statecharts in Java?* – Section 4 presents an *implementation pattern* for mapping behavioral models to Java source code.

Having discussed the modeling and model checking issues and introduced the implementation pattern the final section concludes the paper and presents the directions of our future research.

2 Introducing Exceptions in the Abstract Behavioral Model

In this section we identify the *possible sources* of Java language-level *exceptions*, propose a mechanism for *transforming* them to UML *statechart events* and introduce a *pattern* (statechart design convention) for *handling the events* in the statechart similarly as exceptions are handled in Java programs.

We model event-driven systems by using UML Statecharts. The State Machine package of UML [8] specifies a set of basic concepts (states and transitions) and several advanced features (state hierarchy, orthogonal decomposition, history states etc.) to be used for modeling discrete behavior through finite state-transition systems. The operational semantics is expressed informally by the standard in terms of the operations of a hypothetical machine that implements a statechart specification.

The example discussed in this article is the traffic supervisor system in the crossing of a main road and a country road. The controller provides higher precedence to the main road, i.e., it does not wait until the normal time of switching from red to yellow-red if more than two cars are waiting at the main road (the arrival of a car is indicated by a sensor). Cars running illegally in the crossing during the red signal are detected by a sensor and recorded by a camera. For simplicity reasons only the statechart diagram of the light control of the main road is investigated here (Fig. **1**).

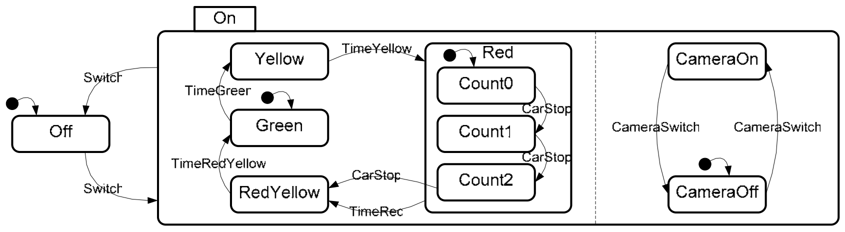


Fig. 1. Statechart of the traffic light example

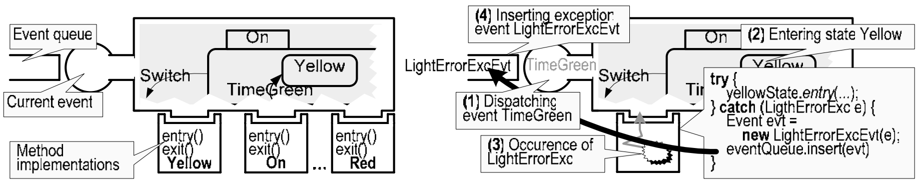
2.1 Reflecting Programming Language-Level Exceptions

The throwable classes (i.e., ones that can be thrown by the *throw* Java keyword and are directly or transitively derived from the *Throwable* class) can be grouped to three main categories: (1) descendants of the *Error* class are thrown by the virtual machine (VM) on encountering a very serious problem (e.g., internal data inconsistency); (2) classes derived from the *Exception* class are used for indicating exceptional situations in the application. Java programs usually throw and catch Exceptions. Since exceptions indicate primarily exceptional situations (i.e., ones that require some handling that is different from the normal execution flow) exceptions should not be considered as fault notifications only: it is completely normal to throw and catch some exceptions (especially interface exceptions) during the execution of a Java program. The only distinguished class directly derived from the *Exception* class is the (3) *RuntimeException* class that represents non-systemic issues (i.e., ones typically resulting from application bugs like accessing objects by null reference) detected by the VM.

2.2 Transforming Java Exceptions to UML Statechart Events

Our goal is to support the modeling of exception handling using a modeling pattern based on the standard notation of statecharts. The prerequisite of this approach is to elevate exception handling to the abstraction level of statecharts by *mapping exceptions thrown in actions* (i.e. occurring in the VM or in the routines implemented by the application programmer) *to events* of the statechart. Accordingly, before presenting the modeling pattern, we will introduce this mechanism.

Taking into consideration the implementation of a UML statechart the application code can be divided into two parts: the *event handler engine* that is responsible for coordinating the behavior according to the statechart specification and the *routines implemented by the application programmer* (state entry and exit actions and actions associated to transitions, including all the libraries used by these routines). In this discussion and in the implementation pattern proposed at the end of the paper we assume a *callback-style implementation* [9] of the statechart (Fig. **2**, left): actions are methods of Java classes (called *action methods* here) and the event handler engine calls them according to the statechart specification, the configuration and the current event.



```
try {
    yellowState.entry(...);
} catch (LigthErrorExc e) {
    Event evt =
        new LightErrorExcEvt(e);
    eventQueue.insert(evt)
}
```

Fig. 2. Left: Callback-style statechart implementation; right: dispatching an exception thrown by an action method and transforming it into an exception event

In this point of view the *interfaces* of the action methods are the borders between the event handler engine and the programmer-written parts. Obviously the Java exceptions are to be used in the programmer-written routines as in any other applications without modifying the programming model but when reaching the event handler engine–action method interface they are to be caught since the propagation upwards the function call stack would destroy the normal operation of the event handler engine. After catching the exception an appropriate event is to be inserted in the event processing queue. All this mapping is performed in the event handler engine.

For example a safety criterion against the traffic controller can be to achieve a *fail-safe* state after the failure of one of the lights by switching off the power of all the lights in the crossing. The fault is detected by the entry actions of specific states, e.g., the entry action of the Yellow state detects a short circuit, and the appropriate code section throws a *LightErrorExc* exception. This way when the event handler engine catches an exception after calling the function implementing the entry action of state Yellow it inserts a *LightErrorExcEvt exception event* in the queue (Fig. **2**, right)

This approach necessitates the discussion of two issues: (1) the transformation of exceptions to events should not interfere with the run-to-completion (RTC) semantics of the UML statechart specification and (2) the insertion of exception events in the queue should be compatible with the event queue semantics:

- The RTC semantics of UML statecharts means that an event can only be dequeued and dispatched if the processing of the previous event is fully completed. Our approach meets this requirement since (i) the insertion of a new event in the event queue does not interrupt the processing of the current event and (ii) actions can be considered to be performed even if they throw an exception since the action methods can provide a cleanup routine in a finally block if necessary.
- The UML standard does not restrict the dequeuing policy of the event dispatcher leaving open the possibility of modeling different priority schemes; in this point of view inserting an exception event in the head of the processing queue is an implementation of a priority scheme that provides high priority to exception events.

2.3 The Exception-Event Handling Pattern

The following discussion introduces a modeling pattern (statechart organization pattern) that enables the handling of exception events in a similar fashion as Java language exceptions are handled in programs. The key concepts of the exception model in Java to be implemented in UML statecharts are as follows: (1) exception handling routines are *separated* from the regular code; (2) exceptions may be *propagated* upwards the control hierarchy until an appropriate handler is found and (3) groups of exceptions can be established by organizing them in a *class refinement hierarchy*.

The exception event handling pattern introduces composite states similar to the *try-catch-finally* blocks of Java programs. The *TryBlock* composite state encloses the normal operation to be carried out without having to handle exceptional situations (Fig. **3**). The *CatchBlock* composite state contains substates that enclose the activity to be performed when handling specific exceptional situations. Transitions triggered by the appropriate exception events connect the *TryBlock* composite state to corresponding substates of the *CatchBlock* composite state. The *FinallyBlock* composite state encloses the activities to be performed after the *TryBlock* regardless whether an exception event has been delivered or not; this is why transitions with empty triggers connect the *TryBlock* and *CatchBlock* composite states to *FinallyBlock.*

```
try {
  tryOp1;
  if (pred)
    tryOp2;
  else
    tryOp3;
  tryOp4;
} catch (Exc1 e) {
  catch1op1;
  catch1op2;
} catch (Exc2 e) {
  catch2op1,
} finally {
  finallyOp1;
}
```

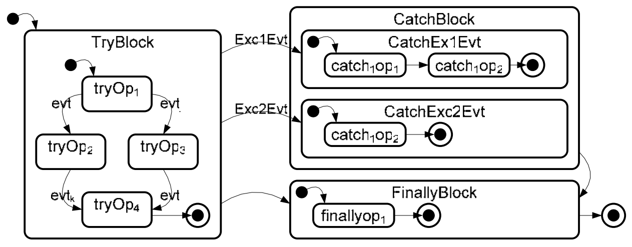


Fig. 3. The exception-event handling pattern

On the left side of Fig. **3** a Java pseudo source fragment is presented while on the right side a UML statechart that performs *similar behavior* as the Java program (note that the source is not the implementation of the statechart, just an analogy). It is easy to see that following this pattern the behavior of the statechart will be analogous to the Java exception handling semantics and provides the same key benefits:

- Since the exception handling routines are moved into composite states the *separation from the normal behavior* is even more visual than the catch blocks in Java.
- Since states can be refined to any depth, the *propagation* of Java exceptions upwards the function call stack is analogous to recursively applying the pattern (e.g., refining the state *tryOp$_1$* in Fig. **3** in a similar fashion). If an exception event occurs, it will trigger the transition at the deepest level of the hierarchy where a corresponding *Catch* state is defined (as the virtual machine invokes the catch block that corresponds to the function at the deepest level of the function call stack interested in handling the exception). In Fig. **4** the *Called* composite state is aware of only handling *SpecExcEvt* but since it is embedded in a structure that catches the more generic *GenExcEvt* exception event class, if a *GenExcEvt* exception event occurs in *Called*, the event triggers the transition at the higher hierarchy level and the execution continues with the appropriate catch block.

```
void called() throws GenExc {
  try {
    tryop1;
    tryop2;
  } catch (SpecExc e) {
    catch1op1;
  }
}
void caller() {
  try {
    called();
  } catch (GenExc e) {
    catch2op1;
  }
}
```

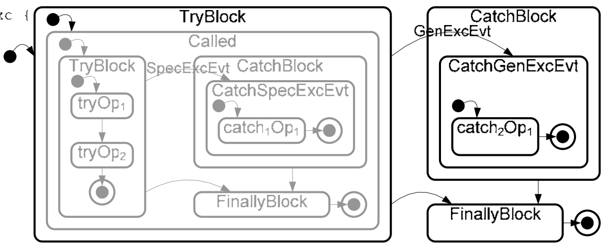


Fig. 4. Propagation of exception events

- Exception events can be organized into a class hierarchy similarly to Java exceptions. A restriction of our approach is that the order of catch blocks can not be represented in the statechart. This way one can not handle a generic exception event and an event derived from it at the same level of the statechart since according to the statechart semantics both transitions triggered by the exception events would be enabled resulting in a non-deterministic selection.

The statechart of the traffic lamp (Fig. **1**) after introducing *LightErrorExcEvt exception event* as the statechart-level representation of the *LightErrorExc* exception according to the exception-event handling pattern is shown in Fig. **5**.

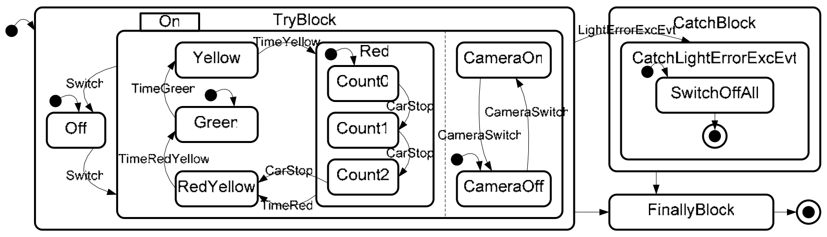


Fig. 5. Statechart of the traffic lamp after applying the exception-event handling pattern

3 Model-Checking Exception Handling

The Extended Hierarchical Automaton (EHA) notation is a well-elaborated representation of finite state-transition systems in a clear structure with formally defined semantics enabling model checking (e.g., analysis of reachability, safety and liveness properties). Since EHA are capable of representing the key modeling constructs of statecharts (state hierarchy, concurrent decomposition, interlevel transitions etc.), automatic transformations were proposed for mapping UML statecharts to EHA [11]. The *syntax* and *semantics* of EHA is described in [2]. In the following a short informal overview is given mainly focusing on the representation of UML concepts.

- An EHA consists of *sequential automata* which contain simple (non-composite) *states* and *transitions*. These states represent simple and composite states of the UML model. States can be *refined* to any number of concurrently operating sequential automata. Composite states of the statechart 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. 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. A transition is *enabled* if its source and all states in the source restriction set are active, the actual event satisfies the trigger and the guard is enabled. On taking the transition the target and all states in the target determination set are entered.

The EHA representation of the traffic light example after applying the exception event handling pattern (Fig. **5**) is shown in Fig. **6**. States of the statechart are mapped to EHA states. Concurrent and non-concurrent refinement is expressed by automata refining the appropriate states (the refinement is expressed by grey arrows in Fig. **6**).

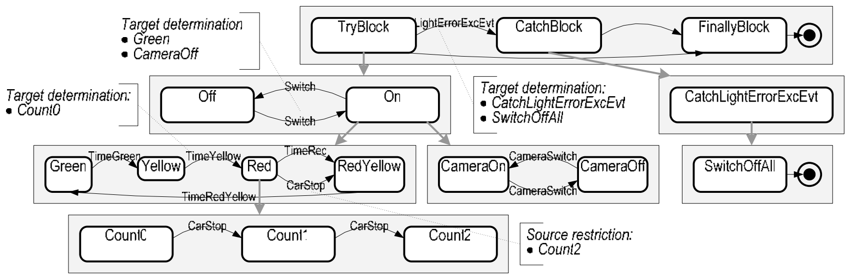


Fig. 6. Extended Hierarchical Automaton of the traffic light example

The metamodel (graphical syntax) of EHA is presented in Fig. **7**

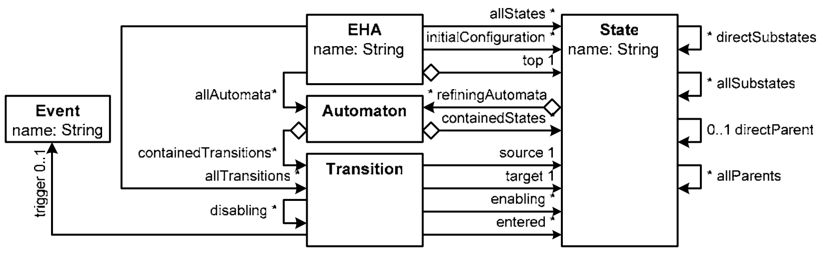


Fig. 7. Metamodel of Extended Hierarchical Automata

The EHA representation, as an abstract syntax of UML statecharts, is a basis of automatic model transformation to model checker tools [2]. Since our approach for introducing exceptions into statecharts does not modify either the statechart syntax or the semantics, only two issues are to be solved for enabling the model checking of exception-aware statecharts by the EHA-based legacy model checkers.

The first problem is to decide how to represent the hierarchy of exception events in EHA models. One solution could be to add the generalizability feature to the EHA Event concept by introducing a self-association of the Event metaclass in the metamodel, but this modification would require the corresponding modification of the EHA semantics and re-implementation of the EHA-based model checkers. In order to support legacy model checkers our approach is simply the following: if an event class

E is refined to event classes R_1, R_2, …,R_N, substitute all transitions triggered by E with a set of transitions triggered by the non-abstract event classes from the {E, R_1, R_2, …,R_N} set. This way the only modification needed is to implement syntactic substitutions in the tools implementing the UML statechart–EHA transformation.

The second problem is, that from the point of view of checking the behavior in exceptional situations the UML statechart model is open, since the sources of the events representing RuntimeExceptions (thrown by the Java virtual machine) and checked exceptions (thrown by routines written by the programmer) are missing. Accordingly, the model is to be closed by the verifier attaching a model of the run-time environment that generates exception events. (This step is required for model checking only.) In case of exhaustive analysis (i.e., when all possible interleaving of exceptions with normal events is considered) the generation of the model extension is a systematic process of inserting concurrent regions (where exception events are generated) to TryBlocks (where the exceptions are to be caught). The automatic implementation of this extension is a task of our current work.

4 Automatic Generation of the High-Level Behavior

This section presents our code generation pattern (Fig. **8**) proposed for automatic implementation of behavioral models specified by EHA. Our goal was not only to present an approach that is capable of mapping the EHA behavior to the Java language but we were concerned in reducing the impact of the code generation pattern on the programming model followed by the developer. The pattern does not require the implementation of any interfaces or deriving active application classes from any specific base classes. This is achieved by detaching the implementation of the EHA-based behavior from the active application classes in the following way.

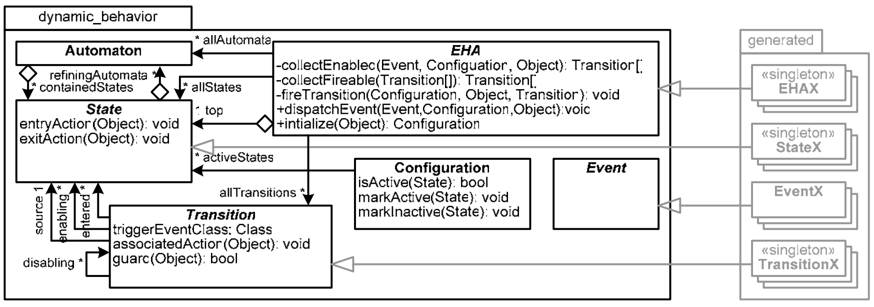


Fig. 8. Classes of the implementation pattern

The *dynamic behavior* is enclosed within stateless classes that are implementations of specific EHA (i.e., they contain the corresponding structure information and all the callback functions implementing the actions etc.). The actual application class (e.g., the *TrafficLamp* class in case of the example) is considered as the *context* of the behavior, i.e., the application class is provided as a function parameter for the behavioral classes for accessing application-specific variables etc. This approach also necessitates active application classes to explicitly store the state configuration information since the classes implementing the behavior are stateless.

The pattern of implementing the event handler engine is essentially an interpreter consisting of two parts: the base support classes are in the *dynamic_behavior* package while the classes to be automatically generated are in the *generated* package.

From a *structural* point of view (how to represent EHA in Java) the classes *EHA*, *Automaton*, *State*, *Transition* and *Event* are the Java equivalents of the metaclasses of the EHA metamodel i.e., their instances and the associations between them are the mapping of the metamodel to the Java language.

From the point of view of the *application dynamics* (how to insert Java code into actions guards etc.), the methods of these classes are the points where the abstract concepts (actions, guards etc.) are to be filled with concrete programming language-level implementation: state entry and exit actions are methods of the *State* class (*State.entryAction, State.exitAction*) while the action associated to the transition and the guard predicate are methods of the *Transition* class (*Transition.associatedAction, Transition.guard*). Since actions and guards are specific to the concrete state and transition, these classes are abstract: their methods are to be implemented in the derived (generated) classes. The current state configuration is represented by the *Configuration* class that maintains associations to the currently active states. All active states exposing EHA-based behavior should contain an instance of this class.

The interpreter is implemented by the *EHA* class. The entry points are the *dispatchEvent* and *initialize* methods. As discussed above the active application classes acting as the context of the operation are provided as generic (*Object*) parameters to the methods. The *initialize* method is used for initializing the configuration of the active object (i.e., taking the initial transition in the UML statechart terminology). The real event dispatcher is implemented by the *dispatchEvent* function. This function is called by the application class passing the event, the configuration and the reference to itself (i.e., to the application class acting as the context of the behavior) as function arguments. The *dispatchEvent* function first calls the *collectEnabled* function that traverses the association to the *Transition* instances (role *allTransitions*) and collects the enabled transitions, then removes from this set the transitions that are disabled by priority relations (*Transition.disabling* association) by calling the *collectFireable* method. The transitions selected for firing are performed during the execution of the corresponding calls to the *fireTransition* method; this involves (1) performing the state exit actions (i.e., calling the *State.exit* methods of states enumerated by the *source* association), (2) performing the action associated to the transition, (3) performing the state entry actions (i.e., calling the *State.entry* actions of states enumerated by the *entered* association), finally (4) updating the configuration.

The interpreter catches the possibly occurring exceptions in the programmer-implemented functions and transforms them to exception events. The language-specific constructs (i.e., try-catch blocks in Java) are used here to enclose these functions (see the upper left box with source code fragment in Fig. **2** right)

The interpreter functions do not need any modification with respect to the actual behavioral model. The really critical and error-prone parts (setting up the object structure expressing the EHA) are automatically created by the code generator.

5 Summary and Future Work

Our paper has proposed a framework that supports the entire development chain of programs that exploit the benefits of exception handling. The modeling of exceptional situations is introduced to statecharts by converting the possibly occurring exceptions to events and reacting to these events in a similar fashion like Java exceptions (i.e., try-catch-finally constructs) by organizing the statechart according to our proposed pattern. Since our approach does not require the modification of the statechart semantics, legacy model checkers can be used for checking the behavior of the system even in presence of exceptions. Finally we presented our implementation pattern for source code-level instantiation of exception-aware statecharts as applied in our prototype code generators. In the near future we would like to apply our exception handling scheme for reacting to behavioral errors detected by our previously published fault detection techniques based on statechart monitoring and run-time checking of temporal logic specification.

References

1. Y. Ahronovitz, M. Huchard. Exceptions in Object Modeling. Finding Exceptions from the Elements of the Static Object Model. In A. Romanovsky et. al (eds): Exception Handling. LNCS 2022, Springer, pp 77-93, 2001.
2. D. Latella, I. Majzik, M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. In Formal Aspects of Computing, 11(6), pp 637-664, Springer, 1999.
3. M. Samek. Practical Statecharts in C/C++. CMP Books, Kansas, USA, 2002.
4. M. Samek, P. Y. Montgomery. State Oriented Programming. Embedded Systems Programming, 2000.
5. J. D. Choi,, D. Grove. M. Hind, V. Sarkar: Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs. In Proc. Program Analysis for Software Tools and Eng., 1999.
6. S. Sinha, M. J. Harrold. Analysis and Testing of Programs with Exception-Handling Constructs. IEEE Trans. on Software Engineering, 26(9), 2000.
7. G. Brat, K. Havelund, S.J. Park, W. Visser. Java PathFinder: Second Generation of a Java Model Checker. In Proc. of CAV Workshop on Advances in Verification, 2000.
8. G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 1999.
9. G. Pintér, I. Majzik. Automatic Code Generation based on Formally Analyzed UML Statechart Models. In Proc. Workshop on Formal Methods for Railway Operation and Control Systems, pp. 45-52, l'Harmattan, Budapest, 2003.
10. C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, F. C. Filho. Exception Handling in the Development of Dependable Component-based Systems. Software Practice and Experience. 2003.
11. D. Varró, G. Varró, A. Pataricza. Checking General Safety Criteria on UML Statecharts. In Lecture Notes in Computer Science, number 2187. Springer Verlag, 46-55. 2003.