PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 46, NO. 1–2, PP. 75–98 (2002)

METHODS OF CHECKING AND USING SAFETY CRITERIA

Zsigmond PAP

Department of Measurement and Information Systems
Budapest University of Technology and Economics
H–1521 Budapest, Hungary
E-mail: papzs@mit.bme.hu

Received: Dec. 18, 2002

Abstract

This article describes methods and tools for automated safety analysis of UML statechart specifications. The general safety criteria described in the literature are reviewed, updated and applied for using in automated specification completeness and consistency analysis of object-oriented specifications. These techniques are proposed and based on OCL expressions, graph transformations and reachability analysis. To help the checking intermediate representations will be introduced. For using these forms, the correctness and completeness of checker methods can be proven. For the non-checkable criteria two constructive methods are proposed. They use design patterns and OCL expressions to enforce observation of the safety criteria. The usability and the rules of using will be also discussed. Three real systems have been checked by using these methods.

Keywords: system safety, specification, completeness, determinism, UML, statechart, design pattern, OCL, graph transformation.

1. Introduction

Nowadays the complexity of the safety-critical computer systems highly enlarges, and it becomes increasingly a difficult task for engineers to specify the systems. Using formal or semi-formal specification and design languages helps the designer to avoid design and coding faults. The powerful model checker and code testing tools can detect modelling and coding errors on the basis of the software specification, but they are not usable to find specification problems. Most of the accidents are caused by computer programs due to specification errors, like incompleteness and inconsistency [16]. Mistakes in the specification are hard to detect and expensive to correct in the late design phases.

Our aims are to develop methods and tools to avoid the errors of completeness and consistency in UML specification and models. We concentrate especially on the behavioural part of UML, the statechart diagrams, and the structure part, namely, the class diagrams. These are the most complex views of specification, especially the statechart diagrams. Most of the errors are likely to occur here.

Our examination is focused mainly on embedded control systems. In these systems, the controller continuously interacts with operators and with the plant by receiving sensor signals as events and activates actuators by actions. UML statechart formalism allows constructing a state-based model of the controller, describes both its internal behaviour and its reaction to external events.

This article is an extended and updated version of the articles presented on DDECS01 [22] and SafeComp 2001 [23] conferences. Section 2 is a short overview of the safety criteria and checking methods proposed in the literature. Sections 3 and 4 describe our work: section 3 outlines in static checking and the reachability analysis, section 4 demonstrates constructive methods helping the designer obey criteria. Practical experience and examples are shown in section 5 . The paper is closed by a short Conclusion.

2. Safety Criteria

N. LEVESON has specified 47 general safety-related criteria for specification of safety-critical software to avoid the typical specification problems [16]. Most of these criteria are based on 3 basic rules: completeness, determinism and consistency [15]. These three criteria define mathematical properties of specifications; others extend these to special cases. Leveson defined criteria for the most important software models and structures.

When a criterion has been specified, object-oriented methods were not used, so all of these aspects are missing. Leveson has specified the system as a single large state machine. Typically, in object-oriented systems several simple machines are interacting with each other.

Additionally, the criteria are specified in a natural language form rather than formally. Although LEVESON has made formal specification languages to allow the checking of the criteria automatically [13], these languages have not been spread out.

Nowadays the object-oriented UML is becoming popular for specification and design of safety-critical software, but neither the criteria can be used directly in object-oriented specifications, nor the UML takes into consideration the safety criteria and the corresponding design philosophy.

2.1. Criteria Groups

The criteria can be divided into three different groups (*Fig. 1* and [23]):

- well checkable criteria, that can be (automatically) checked;
- well observable criteria, that can be observed easily by the designer;
- other criteria.

The three basic criteria specify the properties for the model which are hard to be observed. These properties are defined for all parts of the model and all its combinations, in fact, on the Cartesian-product of the model elements. To obey them manually can be extremely hard in a large and complex system, but by using automatic verification, violations can be easily disclosed. These criteria are hard to observe, but easy to check. 30% of the criteria are similar to this: they need automatic verification. Generally, they are associated with the specification core, which is the general, application-independent part of the specification.

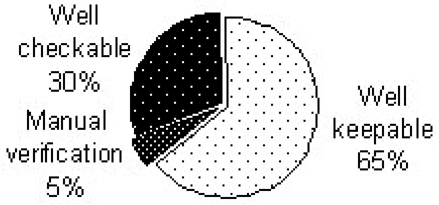


Fig. 1. Classification of the criteria

The other criteria give additional constraints to a specific software structure. They are meaningful and usable only for a specific part of the specification, or for specific software types. The automatic verification of these criteria is difficult.

On the other hand, these criteria are easy to be observed; if the designer is familiar with the software safety and knows the criteria, then he can obey them. Unfortunately in most cases (especially in smaller projects) the designer applies only an incomplete subset of these criteria. Using constructive, non-verification-based methods, the criteria can be observed easily by the designer.

Fig. 1 shows the relation of criteria groups. *Table 1* and *Table 2* show detailed criteria groups for the two most important criteria types. *Table 7* contains additional safety criteria used for designing user interface.

3. Checking the General Safety Criteria

According to the philosophy that has been used by Leveson at the RSML and in other specification languages, a checker program verifies the criteria on the specification (or model). This program detects all incompleteness and inconsistency. After the verification, the designer can correct the errors if it is necessary.

This approach is only usable when both the model for checking and the criteria have adequate formal representations. This means that the model or specification must be defined using a formal method, and the criteria must be formalised, too.

If the model for checking is formalised using statecharts, the criteria must be defined in using the same formalism. The process of the criteria interpretation and adaptation is based on the UML semantics. The details of this can be found in [20]. *Table 1* shows the criteria for checking, and provides a short sample statechart.

Although the UML statechart is based on the same basic formalism [2] as Leveson's specification languages [14], it has several peculiarities. Unfortunately, most of them prevent the verification of the safety criteria. *Fig. 2* shows an example for this. Starting from State1, event e1 can move the machine into any state in the figure, using different statechart constructions. Although some states have priority, in this example the result is unpredictable, since the priority of the transition to State2, State1 and State4 are the same. In addition, some of the usable transitions are not connected directly to State1, so the checker must collect the transitions before verification.

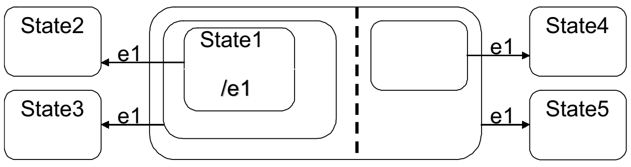


Fig. 2. Mixed construction in the UML statechart

Additional problems with the statechart involve the use of non-formalised guard conditions, and if use of indeterminism (e.g. two transitions can fire from the same state on the same priority level) and incompleteness (e.g. no transition is defined for a state-event pair) are allowed.

Before checking of the criteria, some properties of the statecharts must be modified. The following constraints are necessary [21, 22]:

1. The guard conditions must be restricted into a statically evaluable and checkable form. In this, they can only be logical OR, AND, and NOT expressions of binary terms. These terms are atomic propositions. The checker can evaluate the whole guard condition in every term combination. Note that the guard interpretation results event-cases used by the criteria checker. Details can be found in [22] and [23].
2. There are constructions that are allowed by the statechart, but have a conflict with the safety criteria. They must be avoided (see *Fig.3*):
   a. Two transitions on the same hierarchy level and triggered by the same event start from the same state (indeterminism).
   b. The situation where no transition is defined for a sate-event pair (incompleteness).
   These two properties are in conflict with the two basic safety criteria.
   c. Mixed using of completion and normal transitions.
   Since the event processing gets stuck while the system is in a temporary state, the non-completion transitions cannot fire, so they are unusable.
   d. Jumping out from a concurrent automaton without Join transition.
   Jumping out from a concurrent machine thread, all other threads are finished too. This means, that the construction has side effects. Since the other concurrent threads must be statically complete, the side effect will be conflicted with another transition, and this can be a nondeterministic situation. Although the reachability analysis can find these errors, the use of this construction is dangerous.

e. Deferred events.
Since to every event must be an answer defined in a complete statechart, the deferred event has no meaning.

f. Guard conditions on fork transition output edges or different events of join transition input edges.
This describes illegal statechart constructions (syntactic errors), which are in conflict with the UML semantics too; but some UML tools allow to use these constructions.

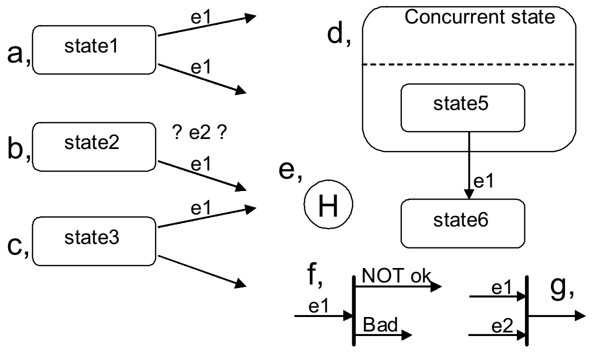


Fig. 3. Constructions of UML statechart should be avoided

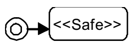
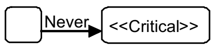
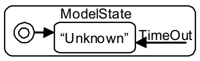
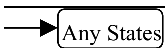
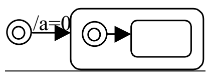
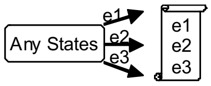
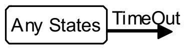
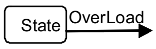
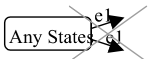
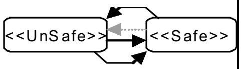
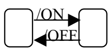
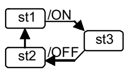
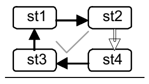
3.1. Static Methods

Essentially the statechart is a highly compact representation of the reachability graph of the program [14]. The guard conditions, the hierarchy and the other special constructions reduce the size and increase the intelligibility of the model. This compression is a source of safety-related errors, since the designer must 'uncompress' the model by checking and strictly adhering to the criteria in his mind. In a complex system this task can be extremely difficult and need the help of a computer program. This program can generate and check the global reachability graph, but it is a very slow and resource-demanding method. Most of criteria allow to skip the construction of the reachability graph, so they can be checked directly on the model. These methods are called static ones.

3.1.1. OCL

It is worth using a language to formalise the criteria, which is part of the UML, such as OCL [18]. The OCL language is designed to express well-formedness criteria,

Table 1. Most important criteria to check with a sample interpretation using state chart

Group	Criteria group	Statechart illustration
State safety	The system should start in a safe state	
	No path to critical states should be included	
	The internal model must be valid	
Completeness	All states must be reachable	
	All variables must be initialized	
	The specification must be complete	
	Timeout transitions must be defined	
	Behaviour must be specified in case of overloading	
	The specification must be deterministic	
	Paths between safe and unsafe states (soft and hard failure modes)	
Output actions and action loops	The output actions must be reversible	
	Repeatable actions must be in live control loops	
	Control loops must be live and checked	

and some CASE tools can interpret and evaluate it. (For instance Argo UML has built-in OCL support, but there are also stand alone OCL interpreters.)

The OCL language is hard to learn and the expressions may be extremely

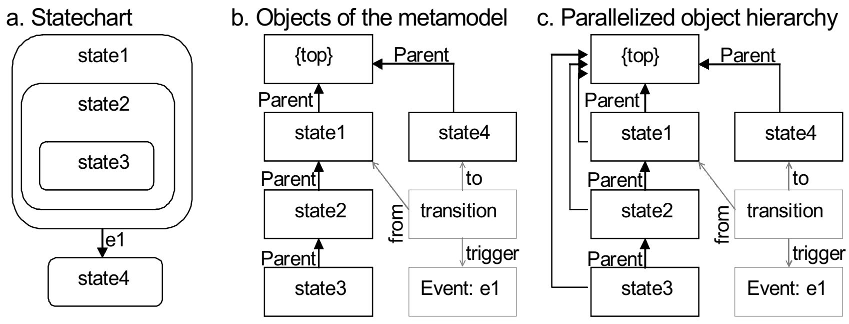


Fig. 4. Sample statechart model (a) and its instantiated object diagram (b) according to the metamodel. Part c shows a modified version to allow OCL checking.

complex, especially in the case of safety criteria. On the other hand, the main problem with OCL language is the statechart metamodel to work on, since it uses recursive structures. If a composite state has sub-states, which has sub-sub-states, the object diagram generated from the metamodel will contain a chain of 'state'-type objects. The length of this chain is not limited. If a criterion wants to collect information for example about inherited transitions, it must navigate through the chain. *Fig. 4* shows such a situation: state3 inherits the transition starting from state1, but to find out this, the checker must navigate trough two links and the object instance of state2.

OCL can handle multiple (parallel) associations, but this is not usable for recursive structures, which are realized on the metamodel level as object chains. By tying model modification or temporary variables, the problem could be solved (for example as *Fig. 4* (c) shows); but OCL cannot allow these.

The problems can be solved by the modification of the metamodel. After removing the chained object system from the metamodel (*Fig. 4* (c)), the OCL expressions could be usable. This can be done by a transformation process.

3.1.2. Graph Transformation

Since the statechart models are eventually graphs, their modification is a graph transformation, which has a sound methodology [12]. The graph transformation is based on pattern matching, which can find a sub-graph pattern in a large graph. If it succeeds, the transformation program can replace it with another piece of graph. After finding all patterns that were looking for and transforming all into the requested form, the graph transformation is ready. As it is obvious, the graph transformation is controlled by two patterns: the one that is looked for and the one by which it will be replaced. They constitute a graph transformation rule (see *Fig. 5*).

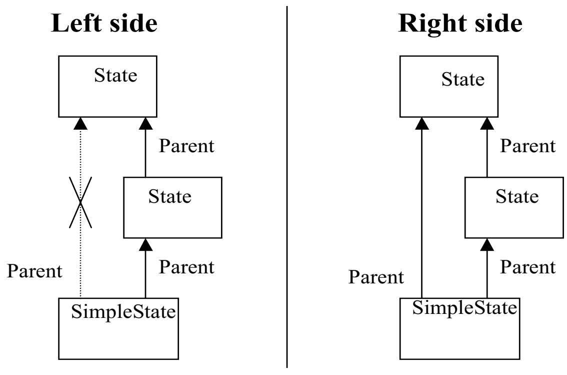


Fig. 5. Example on graph transformation rule to convert the chained structure into a parallel one. (The left side matches only, if the three nodes and the two Parent associations exist, and there is no stroked association.)

By using this transformation method, the complex task of criteria checking can be divided into simple steps. Since we use a formal graph grammar, the correctness of the steps can be proven [23].

3.1.3. Intermediate Representations

The statechart has a large set of usable model elements and constructions. Using these, for example, the hierarchy, concurrency, entry/exit actions, pseudo-states (conditional, fork, join, dot, sync) and the other special statechart elements, the size of model can be highly reduced, but makes formal version of the safety criteria very large and complex. Essentially some of them cannot be directly formalised on the statechart (for example, the rules need the safety classification of the states).

Using graph transformation, the object chains and the complex elements and construction can be converted into a more simple form; to an intermediate representation. The transformation of the most important elements is shown in *Fig.*6.

The conditional transitions can be replaced with separated normal transitions. This makes these transitions checkable together with the other transitions. The Fork transition can be converted into special, multi-output transitions. This process needs the transformation of the sub-start states too. From the point of view of checking the Join transitions are similar to normal transitions with additional checking. The completion transitions are transformed into normal transitions; the transformation program exchanges it with a transition for all trigger events. In this approach the completion transitions work similar to the transitions triggered by all events. According to the UML semantics the event processing is halted in the temporary states, so the transformed version should work differently from the original model.

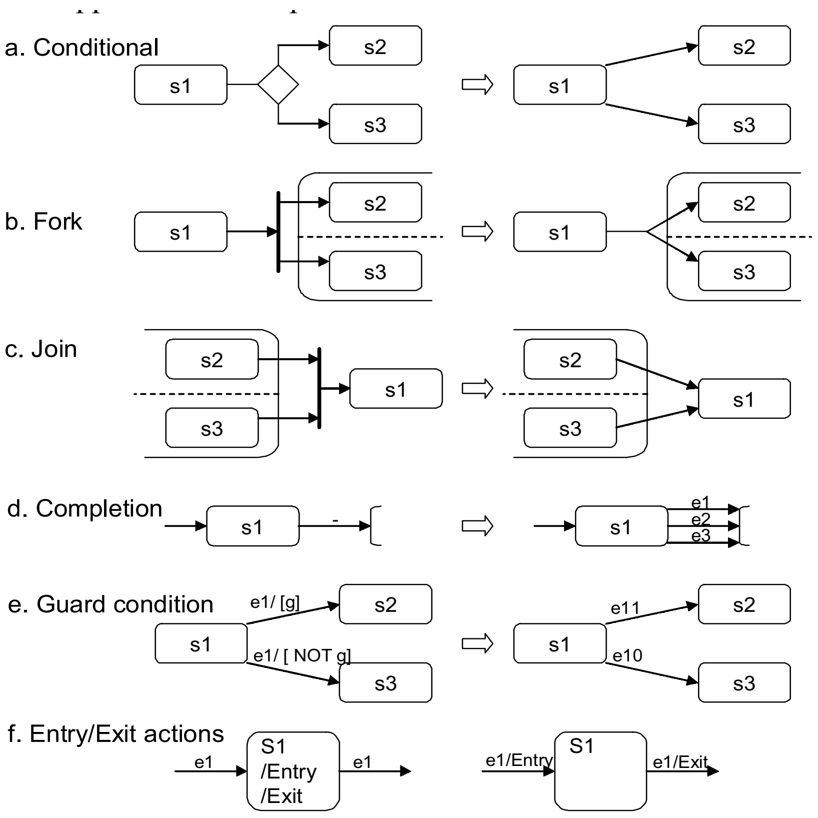


Fig. 6. Transformation of the main statechart elements

Although this means that the transformed model and the original one are not the same, this difference has no effect on the criteria checking.

The guard conditions can be transformed into sub-graphs or states, this intermediate form converts them into sub-events. This approach handles the trigger event and the guard condition together, and interprets them as a special 'event-when-the-guard-is-enabled' super-event. These are the special cases of the original events constructed from the truth table of the guards. The entry condition can be moved to the incoming transitions, the exit actions can be moved into the outgoing transitions. These methods are usable only, if the direct jumping out from the concurrent machine thread is disabled.

The intermediate representation, the Reduced Form [23] consists only of the basic model elements: states, transitions, events, and actions. This form is flat, the hierarchy and concurrency information is transformed into separated representations. The Reduced Form has no Entry/Exit actions, temporary states, completion transitions and internal events. The form is orthogonal, since the model elements are unable to realise the functionality of each other.

Table 2 shows the criteria groups for the available methods for the checking.

Table 2. Summary of available checking methods for well checkable criteria

Criteria group	Reduced form	Reachability analysis	Complexity of manual checking
The system should start in a safe state	Yes	Yes	Medium
The internal model must be valid	Yes	Yes	Low
All variables must be initialised	Statechart only	No	Medium
The specification must be complete	Yes	Yes	High
The specification must be deterministic	Yes	Yes	High
Timeout transitions must be defined	Yes	Yes	High
No path to critical states should be included	No	Yes	High
A behaviour must be specified in the case of overloading	Basics only	Yes	High
All states must be reachable	Static check only	Yes	High
Paths between safe and unsafe states (soft and hard failure modes)	Yes	Yes	High
Repeatable actions must be in live control loops	Loop check only	Yes	Low
The output actions must be reversible	Yes	Yes	Low
Control loops must be live and checked	No	Yes	High

3.2. *Reachability Analysis*

There are criteria and constructions which need the building and checking of the global reachability graph. *Table* 3 shows some of these.

To analyse the reachability graph there are off-the-shelf model checker tools, for example, the model checker SPIN [11]. They have specific input forms; the SPIN uses the Promela language. The UML model or specification must be converted into such a form first; moreover the safety criteria we want to check need conversion, too.

The original statechart model can be easily converted into an extended final state machine [3], using graph transformation. This form is used for the generation of Promela code. If the system contains more than one class and more than one

Table 3. The most important rules need reachability analysis

Construction or property	Safety rule
'in_state' guard conditions	The two (or more) states associated to the construction could be active at the same time.
Enabling join transitions	
Verification of synchrony state	The synchrony state cannot be avoided
Parallel actions in concurrent regions	Two actions cannot start in concurrent regions triggered by the same event.
State and transition reachability	All states must be reachable, except the catastrophic states and all transitions must be used.
Invalid END states	The program cannot halt in a state, which is not an end state, and can halt in all End States.
Infinite loops and deadlocks	The program cannot step into a disallowed infinite loop or dead lock.
Irreversible actions	All safety-critical actions must be reversible.
Action loops	The actions that must be done cyclically should be in a loop.
Virtual inconsistency at the user interfaces	The user must look at the user interface in a consistent way.

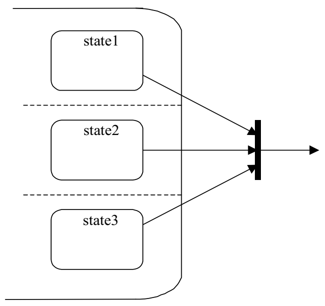


Fig. 7. Sample Join transition; the LTL expression 'EF(state1 & state2 & state3)' must be true

statecharts, the reachability analysis needs the inter-statechart information, too. Because of this, the complete model must be transformed into Promela code, including all statecharts, and their communications, too [1].

In the second phase, the criteria must be converted into formal expressions. SPIN can detect general reachability problems (such as deadlocks) automatically; the other criteria should be converted into Linear Temporal Logic form. Since the content of this form is strongly associated with the model, the criteria are not convertible generally: the static checker program must generate them during the verification of the other criteria.

Fig. 7 shows an example for this. The join transition is enabled only if all source states (State 1, 2, 3) are active at the same time. To verify this, an LTL expression could be generated, which will be verified by SPIN: EF(state1 & state2 & state3).

4. Constructive Methods

65% of the safety criteria are not checkable, but easy to obey. If the checking of the model is not available or inefficient, another approach is promising: in helping the designer to observe the criteria.

4.1. Classification of the Criteria

In this criteria group there are sub-groups: the group of structural and the group of parametric criteria. The structural criteria give restrictions on the structure of the model, while the parametric criteria specify properties of the operational parameters, timings or values. (Using appropriate structural or behavioural diagrams, like class diagram or statechart, the structural criteria are more checkable.)

According to another approach, there are local and global criteria. The range of a local criterion is one model element, especially one of its methods or properties. The global criteria give rules to the association and interaction of two or more model elements. The checking of the global criteria is easier, but the local criteria are more suitable for the constructive methods.

The third approach of classification defines conditional and unconditional criteria. The unconditional criteria must be always true for a type of model element; the conditional criteria specify something for a special case or structure.

Table 4 shows the most important criteria groups and their classification. These criteria are not associated with the specification core, so they have application-specific parts.

4.2. Safe Class Set

The first solution for observing the criteria is to generate a set of well-defined and safe class set to be used by the designer. This class set can contain all necessary elements and aspects that should be used in safety-critical software. If the designer uses this set, the safety criteria will be obeyed automatically.

To avoid incorrect or incomplete using of this class system, well-formedness OCL rules must be associated with the classes. The UML CASE tool can verify them, thus ensures to observe the safety criteria.

The class set shown in *Fig.* 9 is based on the elements defined by criteria such as ‘Action’, ‘Action sequence’, etc. These can be modelled as classes. (There are other model elements, which are not named, but necessary to implement the criteria.) For instance, the criterion 'If an action of an action sequence must be cancelled, the full action sequence must be cancelled too' defines a model element which can process the actions and classifies the unprocessable actions. This means that this criterion refers to the classes and associations in *Fig.* 8. A complete structure, including the most important criteria is shown in *Fig.* 9. This structure contains some parts of the user interface, the data model, the checked inputs/outputs and the controller.

Table 4. Classification of the out of core safety criteria

Criteria group	Type	Global	Cond
Validation and verification of incoming and outgoing values in time and value domains	Parametric	No	No
Exceptions and non-normal operation modes	Structural	No	Yes
Interrupt-driven systems	Structural+ Parametric	No	Yes
Overloading and degradation	Structural	Yes	No
Overloading of the user interface and the human operator	Parametric	No	No
Refreshing and clearing of the data of user interfaces	Structural	No	No
Time stamp	Structural	No	No
Actions, action sequences, its pre-emptivity and cancellation	Structural	No	Yes
Reversible and repeatable actions	Structural	Yes	Yes
Data inconsistency	Structural	No	No
Loopback in the sensors and actuators	Structural	No	Yes
Other application-specific safety criteria	Structural+ Parametric		

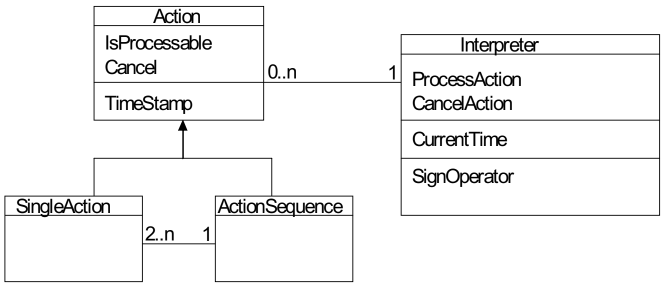


Fig. 8. Example classes of safe class set and its associations

The well-formedness OCL expressions check whether the designer applies this class set properly, and implements all the necessary methods, inheritance and associations. If not, the checker generates a warning message. (In this case a warning message means that the correctness of the model is not guaranteed.)

Building a program from this class set will allow to obey some safety criteria. Naturally, the basic criteria such as completeness must be verified in this model too.

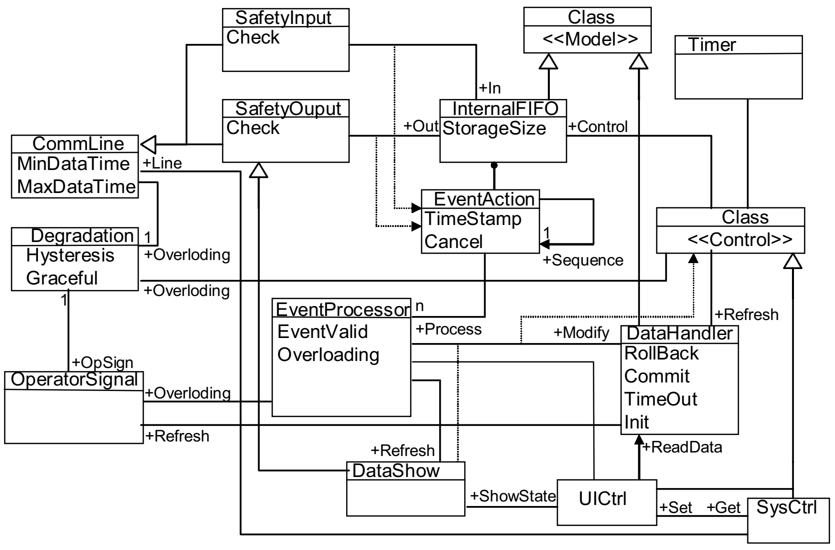


Fig. 9. Simplified version of a safe class set optimised for embedded process control software

4.3. *Structure Patterns*

Unfortunately, it is not easy to use the safe class set, especially for a designer without safety qualifications. To help and speed up the software engineering, it is worth creating some example structures for using this system (*Figs.*9 and 12). The examples are formalised as design patterns.

4.3.1. *Design Patterns*

The design patterns [7] were originally used in the architecture design. The main goal was to help reusing the well-tried models and structures. Actually the design patterns give general solutions to a problem in a well-documented form.

Later on, the design pattern technique started to be used in other areas too. Since code reuse is one of the goals of the object-oriented methodology, this methodology is well fitted to use patterns.

Nowadays a large number of design patterns exist in the area of software engineering. Some of them are using UML, but there are more and more strong efforts to put the design patterns on a fully formal basis instead of semi-formal languages. (For example, the LePUS language is designed to replace the natural language parts with formal diagrams and codes [8]).

There is a large set of design patterns in the area of software safety. They give well-structured and well-tried models to the designer, but concentrate on the software structures, reliability, and self-testing, rather than on the specification completeness and consistency.

Using a design pattern, the software designer inherits a large set of methodology and safety rules, without knowing this.

4.3.2. Using Design Patterns to Observe Safety Criteria

Some criteria recommend a structure for special application areas. For example 'if it is possible, every output of the system must be observed by an input' to verify the correct operation of the output. This means loops in the environment of the software (e.g. in the hardware). Only in the basis of the UML specification is impossible to decide whether the designer has observed the criteria or not. It is the designer's responsibility, but we can help its work by giving predefined software structure design patterns. Although, the correct use of these patterns is not checkable, the probability of the mistakes is reduced.

This method is well usable in the case of the global or conditional criteria. Using it together with the safe class sets and criteria verification, the efficiency is higher. *Table* 5 shows typical application areas of this method. The table gives the most important and most typical tasks and constraints for each structure.

5. Practical Experience and Examples

5.1. Tool Implementation

The techniques described in this paper have been used in three real projects. All of them are safety-related: a train-controller software, an embedded controller of a dialysis machine, and a fire alarm system. The train-controller software includes communication between two large parts: the communication controller and the user interface. The dialysis machine has a distributed structure: it has several simple classes (object) without user interface. Finally, the fire alarm centre has 3 basic parts: the user interface, the data model and the communication controller. Each object has been specified and designed by using UML.

Table 5. Typical areas for using safe structure patterns

General application structures	Typical specification tasks
Structure of interrupt-driven (and time-critical) systems	Minimum and maximum number of interrupts, handling delay
Error detection using external loopbacks	Correct and erroneous answers, stability
Structure and operation of user interfaces	CANCEL function, default screen, no unexpected actions and context-changing, all internal variables must be observable, no side-effects.
Data handling, data protection, and transactions	Obsolete and invalid information, change-sensing, avoiding inconsistency, rollback
UML statechart event queue (and other event queues)	Step-completions, FIFO, time stamp, no internal events, priority system
General software structures (from the point of view of safety, reliability, and other aspects)	Redundancy, watch-dog, monitor structures, etc.
Communication structures	Communication lines, multiprocessors, interface declarations, etc.

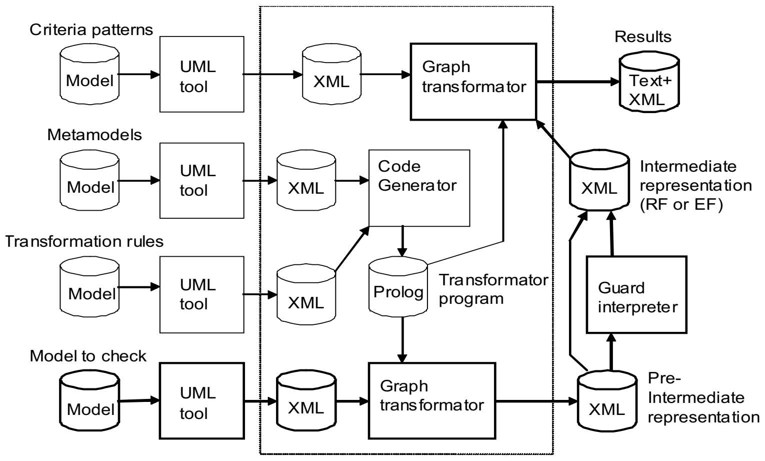


Fig. 10. Transformation and criteria checking method using VIATRA

The modelling tools were Rational Rose and I-Logix Rhapsody. Both tools can export the model into an XMI file, but unfortunately the formats of the two programs are slightly different. This shows that, although the metamodel of the statechart is well specified, the real UML tools are using different versions.

The model transformation program VIATRA [6] has been implemented using SWI Prolog. This system can read graphs given as XMI [17] files.

The rules for the graph transformation are specified in a graphical form, using Rational Rose. There is a special profile defined for designing graph transformation rules using class diagrams. The result is an XML file, which contains the rule description. The VIATRA system reads this file and after a sequence of transformations converts it into a Prolog program: this program will implement the model transformation.

The safety criteria checker consists of a large set of graph transformation rules, a rule control automaton, an external program for the guard condition evaluations, and a text output generator. These parts are used by VIATRA during the transformation. The result is a text file containing the error messages.

The complete procedure is shown in *Fig. 10*.

The basic properties of the checked systems are shown in *Table 6*.

Table 6. Comparison of the system has been checked

Name	Train controller	Dialysis machine	Fire alarm system
Classes	2	28	1
States	47	86	26
Transitions	107	99	118
Hierarchy level	4	6	3
Concurrency level	3	2	3
Events	6	10	4
Guard terms	8	15	6
Number of problems found	87	28	8
Non-trivial problems found	12	20	7

5.2. Design

The structure of the third application, the central control program of a fire alarm system is shown by *Fig. 11*. This program runs on a special microcontroller. In this system the development of the user interface was the main task, but during the project two other modules had to be developed and checked for completeness and determinism.

From the point of view of safety the User Interface of a program is very important. A large number of accidents happen due to some kind of human operator errors. Most of these are associated with a flaw in the specification of the user interface, and would be avoidable if this part of the program were flawless.

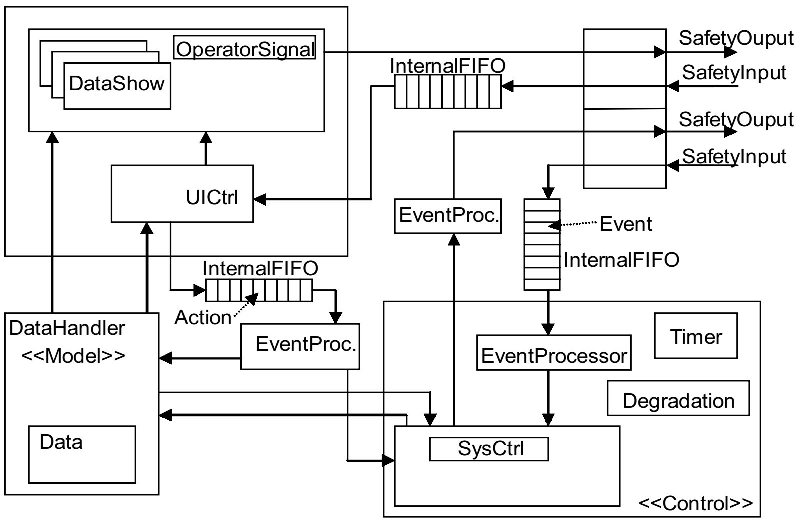


Fig. 11. Simplified block diagram of the sample fire alarm control software, using structure patterns

The fire alarm central program works on an alphanumerical LCD display, and because of the wide functionality of the system, the interface is very complex. Additionally, the special rules of the EN54-02 [9] standard have to be satisfied.

The resulted user interface is based on the Safe UI design pattern (*Fig.* *12*). It has more than 30 Display Data handler objects, including 3 default screens, 4 levels of inheritance and some special display modes, such as 'Display Test'.

5.2.1. *Criteria to Check*

LEVESON has identified typical user interface specification errors, and defined criteria in this area too [4]. This criterion set can be extended with other criteria [19], and with the general ergonomic rules [19]. In addition, the user interface specification must be complete and deterministic, and must realize the other general safety criteria too. (In some cases the field standards can give additional criteria. For instance the EN54-02 [9] specifies special safety rules for fire alarm systems. These special rules must be added to the general safety criteria.) A summary of the criteria is shown in *Table* *7*.

Unfortunately, every program has its own user interface architecture corresponding to the internal structure and the operating system. The range of the user interface types as well as the used methods is very wide. This prevents the use of rigorous rules and checker programs to verify the specification. Although the general criteria (completeness, determinism, etc.) are usable in this area too, the other criteria are put into the group of ‘easy to realize, but hard to check’ and need safe class sets or design patterns.

Table 7. Safety and ergonomic criteria for the user interfaces

Principle	R	S	P
Use of state chart and class model (for UML)	√	√	√
The behavior is fully specified, and consistent.		√	
For all operator events there should be a feedback mechanism	√		
All internal states of the currently handled user interface mode should be displayed to the user (to avoid virtual indeterminism)			√
No side effects allowed	√		
Avoiding the overloading of the operator	√		
When the information is changing the operator should be warned	√		
Automatic data refreshing and clearing mechanism	√		
The operator commands should not refer to external state variables			√
The internal data model must be protected from direct manipulation	√		
The operator commands should be grouped into roll-backable transactions	√		
In every UI state, (except the defaults) should be a ‘Cancel’ function		√	
Shortly after a data-change all operations on the data should be disabled	√		
After a time of no user activity switching to the default screen		√	
No automatic context switching, except in the former case		√	
Every operator event should have a time stamp	√		
Legend: R = realize = the structure performs the rule S = support = the structure allows performing the rule P = possible = the structure does not prohibit performing the rule.			

5.2.2. *Structure of the Safe User Interface*

The most popular user interface structure used in different software is the Model Controller View (MVC) architecture [7]. This model is well flexible, easy to implement, usable in MFC and X-Window environment and in small embedded systems too. This pattern was one of the first design patterns. Nowadays it is used frequently, and this is why the example in *Fig. 12* uses this basic pattern. This is a simplified version of the full Safe UI model [21], which is defined formally, and implemented by using UML.

5.2.3. *Methods of Operation*

In this system the Display Mode handler objects generate the screen data to display. This screen data can be a simple LCD symbol or a complex picture for one or more windows. Generally the screen data is structured into a standard form (for example into a list or a form-page), so the Display Mode handler classes can use inheritance to increase the consistency of the operation and the display structure.

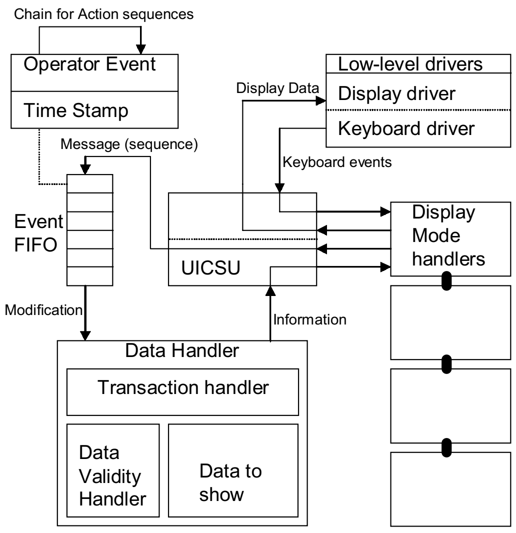


Fig. 12. Sample design pattern for safe user interfaces

To generate the screen data, the Display Mode handler object can read directly the data module to show its information, but has no right to modify the data directly. Only one of these objects has the focus in each moment, this receives the keyboard (or mouse, etc…) events from the user operator. The handling of these events is specified by using a statechart diagram.

If the user gives commands to modify the data in the data module, the currently active Display Mode handler object must send an action or a sequence of actions through a FIFO to the Data Handler unit.

This part of the structure is responsible for the validity of the data. Since more than one software part wants to read or modify the stored information, one aspect of the data handling is the transaction-based operation: modification is only allowed under a strict control. (This is similar to the methods used by database managers.)

If the stored information has changed, the Data Handler unit sends a refresh message to the Display Mode handlers. In some cases the user must be warned explicitly.

If one part of the modification command is bad, unprocessable or obsolete, the whole command or command sequence must be cancelled.

If the information stored in the Data Model is obsolete or invalid, then it cannot be used by the system. It is the responsibility of the Data Handler to identify these situations.

There must be at least one special Display Mode handler object according to an additional rule, which is the default screen. If the user presses the ‘Cancel’ button, a default screen will be activated.

The other details of the safe user interface and its operation can be found in [21].

5.3. *Checking Completeness and Consistency*

Most fire alarm system centrals work as process control systems. In this case the controlled process is the unit that drives detectors rather than group of detectors themselves. The central software must read the detector status information, and control the data collector units, the detector calibration and the fire signal equipment. In addition, all data paths and units must be continuously checked. This is a complex communication task.

In the sample system the communication controller is implemented by a statechart as shown in *Fig. 13*.

The main task in this case was to check the completeness and the determinism. Without this verification the controller software was instable. (On average it committed some fatal communication errors every week and fire protection stopped for some hours.)

During the verification the checker has found some missing transitions and a missing state (drawn by thick lines). After adding the missing components, the communication operated properly.

5.4. *Results*

5.4.1. *Checking the Static Criteria*

A long list of errors and warnings has been generated during checking static criteria. The typical errors are the following:

- No TimeOut transition defined for states

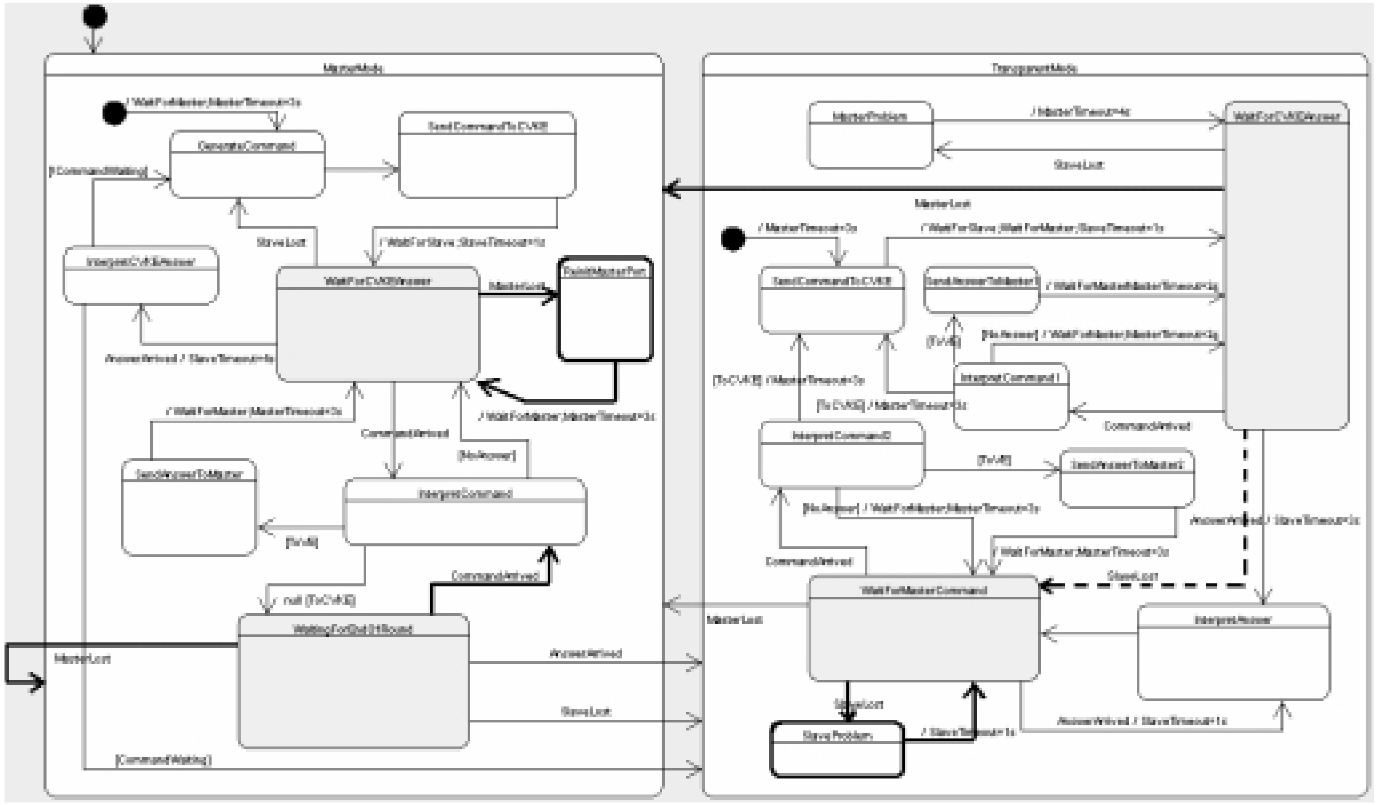


Fig. 13. Statechart of the communication controller

- Incompleteness (caused by case-sensitive guard conditions) or incomplete conditional transitions
- Mixed use of completion- and normal transitions
- Missing initial states or adequate transitions inside composite states
- Invalid guard conditions (using invalid syntax)
- Broken action loops (there is a loop in the graph, where an important action is missing)

Note that 31% of the errors were non-trivial. It required more than half an hour to identify the reason of the message resulted. Without automatic verification these errors could not be detected and avoided.

5.4.2. *Safe User Interface*

The Safe User Interface pattern is associated with 16 safety criteria, (not including the basic ones,) and statically checkable safety criteria. *Fig.14* shows that the safe UI design pattern structurally observes (realizes) 10 criteria from the 16 (62%). Some of them are also checkable on the basis of the Safe Class Set. Other 4 (25%) criteria are supported by the safe UI pattern, but in this case obeying the criteria is the user's responsibility. Although there are two (13%) criteria, for which the pattern guaranties nothing, obeying these criteria is not prevented by the pattern. The detailed information can be found in *Table 7*.

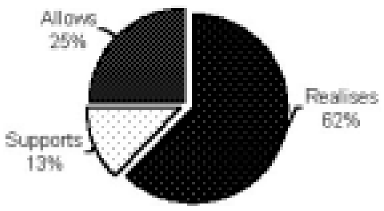


Fig. 14. Efficiency of the design pattern method

6. Conclusion

This paper presented methods and tools for checking of UML statechart specifications of embedded controllers. The existing criteria, which were given in [16], were checked on UML statecharts. Some criteria were checked efficiently by using static methods; others were checked after building the reachability graph of the model.

Since the verification of application-specific criteria cannot be done efficiently, constructive methods are needed to realize the criteria. These techniques such as the use of safe class set or special design patterns help the designer to produce safe software.

References

[1] DARVAS, A. – MAJZIK, I. – BENYÓ, B., Verification of UML Statechart Models of Embedded Systems, In B. Straube et. al. (editors): *Proc. 5th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS)*, April 17–19, Brno, Czech Republic, 2002, pp. 70–77, IEEE Computer Society TTTC – Brno University of Technology, 2002.

[2] HAREL, D., Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, **3**, No. 3, (1987), pp. 231–274.

[3] LATELLA, D. – MAJZIK, I. – MASSINK, M., Towards a Formal Operational Semantics of UML Statechart Diagrams, In P. Ciancarini and R. Gorrieri, editors, *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, Kluwer Academic Publishers (1999).

[4] LATELLA, D. – MAJZIK, I. – MASSINK, M., Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker, *Formal Aspects of Computing*, **11** No. 6, (1999), pp. 637–664, Springer Verlag.

[5] VARRÓ, D. – VARRÓ, G. – PATARICZA, A., Automatic Graph Transformation in System Verification, In *Proc. DDECS-2000*, p. 34, Slovak Academy of Science, 2000.

[6] VARRÓ, D., Automated Program Generation in VIATRA, 2002. Budapest, ISBN 9634206824, pp. 34–36.

[7] GAMMA, E. – HELM, R. – JOHNSON, R. – VLISSIDES, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, Addison-Wesley, Reading Mass. 1994.

[8] EDEN, AH. – HIRSHFELD, Y. – YEHUDAI, A, LePUS – A Declarative Pattern Specification Language, Technical Report 326/98, Department of Computer Science, Tel Aviv University, (1998).

[9] EN-54/2 European Standard for Fire Alarm System Centrals, (1996–1998).

[10] PATERNÓ, F. – SANTORO, C. – FIELD, B., Analysing User Deviations in Interactive Safety-Critical Applications (1998).

[11] HOLZMANN, G., The Model Checker SPIN, *IEEE Transactions on Software Engineering*, **23** (1997), pp. 279–295.

[12] GOGOLLA, M., Graph Transformation on the UML Metamodel, Workshop on Graph Transformation and Visual Modeling Techniques, ICALP'2000, Geneva, Switzerland, 2000.

[13] HEIMDAHL M. P. E. – LEVESON, N. G., Completeness and Consistency Checking of Software Requirements, *IEEE Trans, on Software Engineering*, **22** No. 6, (1996).

[14] LEVESON, N. G. – REESE, J. D. – HEIMDAHL, M., SpecTRM: A CAD System for Digital Automation. Digital Avionics System Conference, Seattle (1998).

[15] LEVESON, N. G. – HEIMDAHL, M. P. E. – HILDRETH, H. – REESE, J. D., Requirements Specification for Process-Control Systems, *IEEE Trans. on SE*, (1994), pp. 684–706.

[16] LEVESON, N. G., Safeware: System Safety and Computers, Addison-Wesley, 1995.

[17] *Object Management Group*, XML Metadata Interchange, 1998.

[18] *Object Management Group: Unified Modeling Language Specification* v 1.3. 1999.

[19] SHNEIDERMAN, B., 1987, 1996, Designing the User Interface, Reading, MA: Addison-Wesley, ISBN 0-201-57286-9.

[20] PAP, ZS., Checking Safety Criteria in UML Statecharts, Technical Report No. 2/2001 of the DMIS, Budapest University of Technology and Economics, 2001.

[21] PAP, ZS. – PETRI, D., *A Design Pattern of the User Interface of Safety-Critical Systems*, IWCIT01, 2001.

[22] PAP, ZS. – MAJZIK, I. – PATARICZA, A. – SZEGI, A., *Completeness and Consistency Analysis of UML Statechart Specifications*, DDECS01, 2001, Győr.

[23] PAP, ZS. – MAJZIK, I. – PATARICZA, A., Checking General Safety Criteria on UML Statecharts, SafeComp2001, Budapest, 2001, pp. 46–55.