Systematically Combining Specifications of Internal and External System Behavior Using Statecharts

Martin Glinz
Department of Informatics, University of Zurich
Winterthurerstrasse 190
CH-8057 Zurich, Switzerland
glinz@ifi.unizh.ch

Abstract

In contemporary model-based specifications, we typically find a naive combination of models of the externally visible behavior of a system (typically expressed as scenarios or use cases) and of the internal system behavior (partially represented in explicit state models and partially expressed as data). However, a systematic combination and integration of the two behavior aspects has not yet been investigated.

In this paper, I sketch a systematic approach for modeling both external and internal behavior of a system with statecharts in an integrated, non-redundant way. The main idea is to start with statecharts that model external behavior in the form of use cases or type scenarios and then add statecharts that model internal behavior only where the scenario/use case statecharts do not suffice for expressing the behavior of the system.

1. Introduction

It is well known that state machines can be used both for modeling system behavior (typically internal behavior of the components of a system) and for modeling interaction between actors and a system (the externally visible behavior or short the external behavior of a system).

In internal behavior modeling, a state represents a situation where the system reacts to those events that trigger one of the state's outgoing state transitions, while the actions specified for the state transitions describe what the system does when a state transition occurs.

When modeling actor-system interaction, every state transition specifies a stimulus coming from an actor and a response by the system. States typically model a time interval where the system waits for the next stimulus from an actor.

Modeling languages such as UML [11], [12] allow both interaction modeling and internal behavior modeling with state machines. However, UML does not care about the methodological aspects of using these models. For example, the state machine of Fig. 1 can be interpreted as a use case, i.e. an external view of the system, as well as a specification of the system's internal behavior.

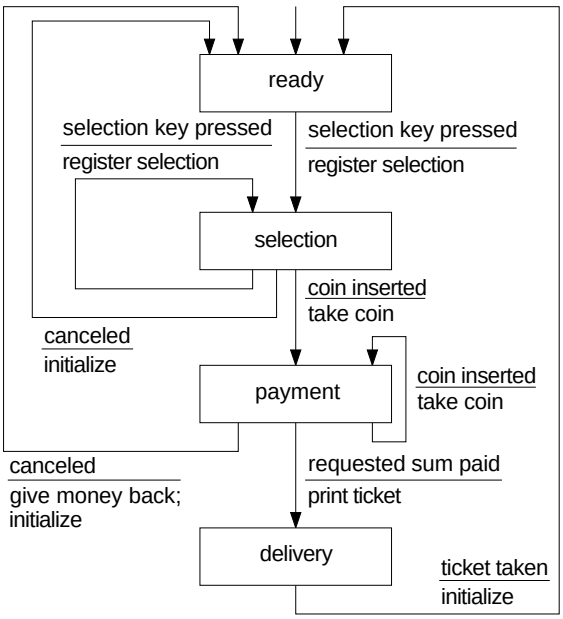

Figure 1. Model of a simple ticket vending machine

In situations where the system behavior is more complex than in the example given in Fig. 1, a pure external behavior specification typically fails, because there is externally visible behavior which depends on results of previous interactions. In this situation, behavior must either be specified by interaction traces (which is not practically feasible in most cases) or a model of internal system states is required.

On the other hand, a purely internal behavior model also fails because it ignores the interaction of the external

actors with the system. Hence, most contemporary specifications combine a model of external behavior (typically in the form of use cases) with a model of internal behavior (both implicitly given by data and explicitly in state machines attached to objects). However, today we do this combination in a naive way, not caring about the relationships and interdependencies between the two kinds of behavior models.

A similar problem arises when modeling components in UML 2.0 [12]. On the one hand, a component in UML 2.0 has interfaces which require a specification of the externally visible behavior of the component. On the other hand, we also have to model the internal behavior of the component, in particular the interaction between its constituent objects and/or inner components.

If we use statecharts both for external and internal behavior modeling, so that formal integration of the two behavior aspects is possible, we come across some questions that—to the best of my knowledge—have not been investigated yet:

- How shall the models be combined so that integration makes sense and has useful semantics?
- Does the combination yield redundant models? If yes, is this redundancy useful (for example, for validation purposes) or is it an unwanted source of potential inconsistencies?
- If we do not want redundant models, how can we achieve that?

In this paper, I sketch a systematic approach for modeling both user-system interaction and system-internal behavior with statecharts [6] in an integrated, non-redundant way. The main idea is to start with statecharts that model type scenarios[1] (or use cases in UML terminology) and then add statecharts that model internal behavior only where the interaction statecharts do not suffice for expressing the behavior of the system.

The rest of the paper is organized as follows: Section two gives an overview of the approach and illustrates it with an example. In Section three, the integration problem is discussed. Section four gives some conclusions, surveys related work and provides directions of further research.

2. The approach

2.1 Basic ideas

The basic idea, which is laid out in this paper, is to have an integrated, non-redundant state machine model of both external and internal behavior of a system, using the external behavior model as the lead model.

This means that external system behavior is modeled first. Internal behavior models are systematically added in all situations where the external behavior models do not suffice, thus leading to a specification where the two behavior aspects are modeled complementary and non-redundantly. In the integrated model, it shall still be clearly visible which state machines model externally visible behavior and which ones represent internal behavior. We achieve this by stereotyping the state machines.

Giving preference to external behavior modeling makes sense because

- in requirements models this is what people are interested in,
- in design models, external behavior modeling supports information hiding and component building.

2.2 Using the tools and materials metaphor

The *tools and materials metaphor* [14] is used for providing guidance which components of a system need which kind of behavior model. This metaphor classifies the components[2] of a system into three categories:

- *'Tools'* assist external actors in doing the actors' tasks: they receive stimuli from them and produce appropriate responses. In order to do their job, 'tools' process 'materials'.
- *'Materials'* represent artifacts (or containers storing artifacts) that are being processed (created, modified, queried,...) by 'tools' or 'automata'. 'Materials' are passive.
- *'Automata'* work without being triggered by an external event. They either run continuously or are triggered internally. Automata are active, processing 'materials' or using 'tools'.

It turns out that external actors interact with 'tools' in this metaphor. 'Automata' act on their own, while 'materials' are used by 'tools' and/or 'automata'. Hence, any component being classified as a 'tool' in the metaphor needs a specification of its external behavior. This specification must be augmented by internal behavior models for all 'materials' that exhibit state-dependent behavior. Finally, all components being classified as 'automata' need a specification of their behavior. As they are parts of the system, we also need an internal behavior model for them.

2.3 An example

As an example, we use a ticket vending system which is more sophisticated than the machine given in Figure 1.

[1] A type scenario is an ordered set of interactions between partners that represents a set of possible interaction steps. A use case is a type scenario.

[2] In requirements specifications, components are identifiable, separable subproblems of the given problem, whereas in design specifications components represent design items with high cohesion and low coupling.

Let us assume that the system consists of four parts:

- Vending is responsible for vending tickets. It is similar to the system specified in Figure 1, but has an additional timeout feature.
- Supervisor supervises the condition of mechanical components such as the coin vault, ticket printer, ticket paper roll, etc.
- Logging logs all vending transactions and all events detected by Supervisor or caused by aborted transactions.
- Maintenance handles offline parametrization and servicing of the machine.

A model of the behavior of this system will be developed in the subsections below.

2.4 The four modeling steps

The proposed modeling method proceeds in four steps:

(1) Classify the components of the system according to the tools and materials metaphor [14],
(2) Model the external behavior of the 'tool' components,
(3) Model the internal behavior of the 'materials' and 'automata' components,
(4) Integrate the resulting statecharts.

The steps are not a strict, waterfall-like sequence, where a step must be completed before the next one can start. Rather they are meant as a sequence of thinking about models: for any piece of model development, modeling external behavior comes first, then internal behavior is considered and the two behavior specifications are integrated.

2.5 The classification step

In the first step, the problem is split into parts and the parts are classified into 'tools', 'materials' and 'automata'. The 'tool' parts interact with external actors or devices and thus exhibit externally visible behavior. The 'materials' and 'automata' parts have system-internal interaction only and, hence, their behavior is internal.

Example. In our example introduced in 2.4 above, Vending and Timeout are classified as 'tools', because they assist the user in accomplishing her/his task, viz. buying tickets. Supervisor is classified as an 'automaton' because it is triggered cyclically by an internal clock. Logging is classified as a 'material', because it mainly represents an artifact, viz. the log of transactions and events. Maintenance again is classified as a 'tool', because it assists the operator in accomplishing her/his task.

2.6 The external behavior modeling step

This step is comparable to classic use case modeling. For every 'tool' component of the system, the type scenarios (in UML terminology: the use cases) are identified. Every type scenario is modeled as a statechart.

Every state transition represents a step in a type scenario, where the triggering event represents the stimulus coming from an external actor, while the triggered actions represent the response by the system. Responses can be (1) messages to an external actor or device, (2) actions that change the state of the system, or both.

In order to simplify integration, all statecharts are modeled so that they are compositional [5].

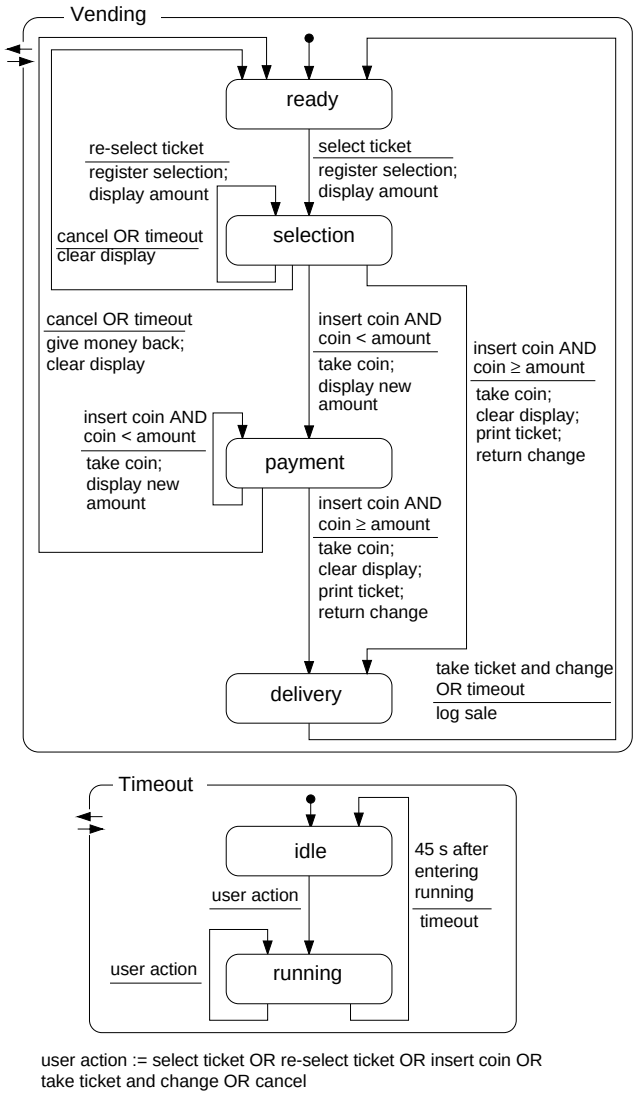

Figure 2. The external behavior of the Vending and Timeout components modeled with scenario statecharts (scenario statecharts are denoted by a double-arrow symbol)

Example. For our example, the behavior models of Vending and Timeout are given in Figure 2. The double arrow symbol in the upper left edge of the statecharts is a

stereotype, marking these statecharts as being type scenarios. The behavior model for Maintenance is omitted here.

2.7 The internal behavior modeling step

In this step, the internal behavior of the 'automata' and 'materials' components is modeled. This step is comparable to classic system behavior modeling with state machines.

The 'automata' components are *active*. Their state transitions are typically triggered by internal events and in turn trigger actions and other internal events.

The 'materials' components, on the other hand, are *passive*. Their statechart is a classic class or object behavior description, specifying the sequences of operations that are allowed on an object. Hence, the events triggering the state transitions are method invocations and there are no explicit actions for these transitions.

Example. Figure 3 gives the models of Supervisor and Logging in our example. The square symbol in the upper left edge of the statecharts is a stereotype, marking these statecharts as being internal behavior specifications.

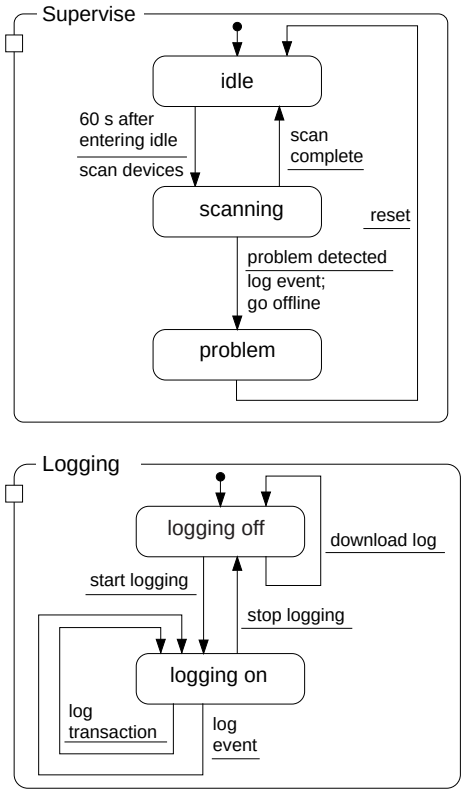

Figure 3. The internal behavior of Supervisor (an 'automaton') and Logging (a 'material') modeled with internal statecharts (internal statecharts are denoted by a square box symbol)

2.8 The model integration step

In the final step, the statecharts that were modeled in the two previous steps are integrated into a single statechart which then is a non-redundant model of the complete behavior of the system. Obviously, this statechart contains the constituent statecharts from steps 2 and 3 as sub-statecharts.

During the integration process, inconsistencies between the parts to be integrated must be detected and resolved. Equally, missing information must be identified and added. Hence, the integration process cannot be automated. It is a *manual, method-guided process*.

In order to describe this process, some general discussion of integration and inconsistency problems is needed first. Therefore, integration is presented in a section of its own below.

3. Integration issues

3.1 Kinds of integration

There is a lot of work dealing with integration and synthesis of models, e.g. [2], [8], [9], [13]. In order to characterize my approach, I give a short classification of the kinds of integration that are typically being used.

a. Stakeholder viewpoint integration. A given problem is modeled from the viewpoints of different stakeholders. The integration problem is to build a single model from these viewpoint models [9].

b. Instance model integration. A set of models covers various instances of a given problem. The integrated model abstracts the instances into a type model. Typical representatives of this kind of integration are state machine synthesis approaches where a set of instance scenarios, each one describing a single behavioral thread, is integrated into a state machine model which represents all possible behaviors [8], [13].

c. Intra-aspect model integration. Some aspect of a system is modeled by a set of non-overlapping partial models. These models are integrated into a single model of the given aspect. The integration approach in [2], where a set of different, but related type scenarios (i.e. use cases) is integrated into a common model, is a typical example of this kind of integration.

d. Cross-aspect model integration. Different aspects of a system are modeled separately. The integration problem is to bring these aspect models together. This is the principal integration problem when modeling with UML. Another example for this kind of integration is the lightweight integration of use cases and class models presented in [3].

In the approach described in this paper, only aspectual model integration (cases c. and d. above) is considered.

Instance model integration (case b.) is no issue here, because all partial behavior models are statechart models, i.e. they already represent classes of behavior, not instances. Stakeholder viewpoint integration is also not treated. It is assumed here that viewpoint integration takes place when creating the statecharts that model the behavior of the system components.

3.2 Dealing with inconsistency

A major problem of any integration approach is dealing with inconsistencies between the models that are to be integrated.

We distinguish inconsistencies in syntax, naming, information flow and semantics.

Syntactic inconsistencies occur when the same concept is represented with different syntax in two constituent models. As an example, assume a model where there are two outgoing flows f and g, while in another model, we have an incoming compound flow consisting of f and g.

Naming inconsistencies occur when the same concept is given different names in two constituent models. The principal means for avoiding naming inconsistencies is using and maintaining a glossary.

Information flow inconsistencies occur when a component B requires information from another component A which is not produced by A. Another, milder form of information flow inconsistency occurs when a component produces an information which is not needed anywhere else.

Semantic inconsistencies occur when two constituent models have a different understanding of the underlying problem. For example, consider a seat reservation system and let the model of the external behavior have a waiting list option for the case where no seats are available. On the other hand, assume that in the model of internal behavior, there is no concept of maintaining a waiting list. In such a situation, we have a semantic inconsistency.

In the approach described in this paper, models with semantic inconsistencies cannot be integrated as they are. In order to make integration possible, semantic inconsistencies must be resolved by modifying the statecharts involved. Nevertheless, the approach also has benefits with respect to semantic inconsistencies: it helps detecting them systematically.

3.3 The integration process

As already mentioned, statechart integration as described in this paper is a manual, method-guided process. Hence, the person doing the integration needs knowledge about the system to be modeled or easy access to the persons having this knowledge. The process consists of two major activities: *syntactic integration* and *detection and resolution of inconsistencies*.

Note that inconsistencies are resolved during integration; the approach does not aim at tolerating inconsistencies [1], [10].

a. Syntactic integration. Syntactic integration is done according to the method given in [2]. We construct a statechart hierarchy having the constituent statecharts as its leafs. Statecharts being independent of each other become refinements of parallel states in this hierarchy. Statecharts with a sequential dependency become refinements of two sequential states, those which mutually exclude each other become refinements of alternative states, etc.

Example. Figure 4 shows the high-level statechart which integrates the previously developed element statecharts. Vending and Timeout are independent, so they become parallel statecharts. Both Vending and Timeout have a sequential dependency on Maintenance (and vice versa). Hence, state transitions from Maintenance to Online and vice versa are introduced. Supervise and Logging are independent of the Maintenance/Online cluster, hence we do a parallel composition here.

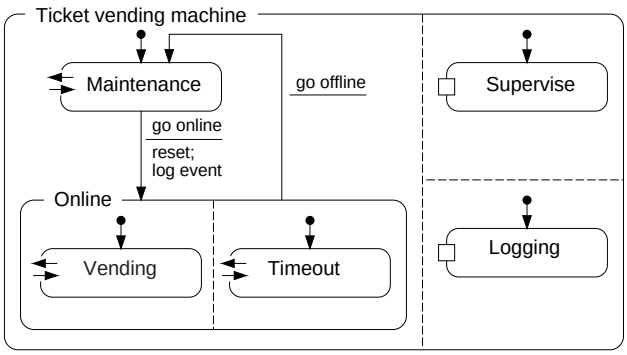

Figure 4. The composite statechart representing the integrated behavior (both external and internal) of the system

b. Detection and resolution of inconsistencies. In order to detect and resolve inconsistencies, all state transitions are inspected, one by one.

We start with the statecharts modeling behavior of 'tools' and 'automata' components. Recall that for all state transitions of these statecharts, the triggered actions are responses (to an external actor or device) or internal actions that cause a change in the state of the system[3]. Now every such transition is inspected by answering two questions: (1) Which actions must be taken by the system

[3] For the sake of simplicity, we regard events that are produced by a state transition also to be (primitive) actions.

to produce the required response? (2) Which actions must be taken by the system to change its state properly?

Next, the statecharts modeling the behavior of 'materials' components are inspected. Here, the question for every transition is: (3) Is the triggering event produced where it should in the statecharts of the 'tools' and/or 'automata' components?

Answering question (1). Let t_i be the transition currently under inspection. For every action A that must be performed by the system to produce a response, identify the component of the system which performs this action and produces the proper response. There are three cases to consider: (a) the action is *local*, i.e. the component containing t_i also performs the action, (b) there exists *another component* X performing the requested action, and (c) there is *no such action*. In case (a) there is nothing to integrate concerning action A. In case (c) we have a semantic inconsistency which must be resolved by modifying the inconsistent statecharts (see 3.2 above). In case (b), identify the state transition t_x in the statechart of component X that triggers action A. Check whether this statechart is currently in a state where transition t_x is enabled. If not, we again have a semantic inconsistency. If yes, check whether the triggering event of t_x is produced by t_i. If this is true, action A is integrated properly. If not, check whether there is a syntactic or naming inconsistency. If yes, resolve the inconsistency by restructuring or renaming. Otherwise we have an information flow inconsistency: The required action and its triggering internal transition t_x exist, but t_i does not produce an event that triggers it. This inconsistency is resolved by adding the missing event to the list of events produced by t_i.

Answering question (2). The process is more or less the same as for answering question (1). The difference is that we do not check for responses, but for proper state changes, i.e. the state changes must be so that scenarios being executed later behave as expected.

Answering question (3). Recall that we now inspect the transitions of the statecharts modeling the behavior of 'materials' components. Let C be the component and t_i the transition currently under inspection. Look at the requirements for C. Identify all transitions $t_1,...,t_n$ in other statecharts which must trigger t_i in order to satisfy the requirements for C. In case of perfect match ($t_1,...,t_n$ all produce an event that triggers t_i), we are done. Otherwise, look for syntax and naming inconsistencies and resolve them. The remaining cases are information flow inconsistencies: an event triggering t_i should be produced by another transition t_j, but it is not. This inconsistency is resolved by adding the missing event.

Example. In our example, the integration process detects and resolves the following inconsistencies. When inspecting the transition from state delivery to state ready in component Vending (Fig. 2), we find that a logging action must be performed. We easily identify the Logging component as being responsible for that action. In this component, there is an action log transaction. However, in the Vending statechart, the action is named log sale. Thus, we have a naming inconsistency which is resolved by renaming the event in the Vending statechart (Figure 5).

When inspecting the Logging component, we find a requirement that aborted transactions are events which must be logged (see Section 2.3). We have an action log event, but it is not triggered in every transition where it should be: in the Vending statechart, we find two transitions where a transaction is aborted, but no logging takes place. Hence, we have an information flow inconsistency: log event must be added to these transitions (Figure 5).

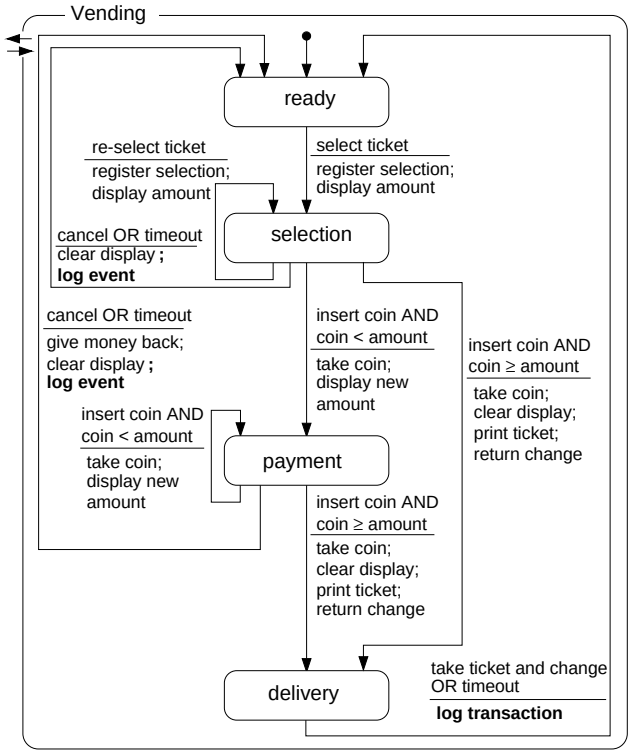

Figure 5. Additions and corrections to the Vending statechart for resolving inconsistencies detected during integration (modified elements in boldface)

4. Conclusions

In this paper I have sketched an idea how to systematically combine specifications of external and internal system behavior on the basis of statechart models.

To the best of my knowledge, there is no previous work on this kind of behavior integration. Existing work on state machine generation/composition from scenarios [8], [13] focuses on transforming a requirements model of

external behavior (typically given by message sequence charts) into state machines, which are typically attached to classes in an architectural model. Viewpoint integration [9] is another type of integration which is not considered here.

SCR [7] uses an integrated behavior model, but in a very specific context: SCR is based on the idea of continuously observing a set of external items (monitored variables) and producing values for another set of external variables (controlled variables). The interaction between the environment and the system is standardized: the system cyclically goes through the monitored variables and produces values for all the controlled variables according to the system behavior model.

The approach described in this paper is not meant as a final solution, but as a starting point for discussion and investigation. In particular, the feasibility and usefulness of this approach have to be validated with real world examples. The problem of incremental specifications has to be addressed. The combination of a behavior model as described in this paper with a model of objects and object structure should be explored. Some work has already been done in my research group [3], [4]. Currently, we are also investigating the impact of such integrated models on executing behavioral system models.

References

[1] Balzer, R. (1991). Tolerating Inconsistency. *Proceedings 13th International Conference on Software Engineering.* 158 - 165.

[2] Glinz, M. (1995). An Integrated Formal Model of Scenarios Based on Statecharts. In W. Schäfer and P. Botella (eds.): *Software Engineering – ESEC'95*. Berlin: Springer. 254-271.

[3] Glinz, M. (2000). A Lightweight Approach to Consistency of Scenarios and Class Models. *Proceedings 4th IEEE International Conference on Requirements Engineering*. Schaumburg, Ill. 49-58.

[4] Glinz, M., S. Berner, S. Joos (2002). Object-oriented modeling with ADORA. *Information Systems* **27**, 6. 425-444.

[5] Glinz, M. (2002). Statecharts for Requirements Specification – As Simple as Possible, as Rich as Needed. *1st ICSE International Workshop on Scenarios and State Machines: Models, Algorithms and Tools*, Orlando.

[6] Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* **8** (1987). 231-274.

[7] Heitmeyer, C. (2002). Software Cost Reduction. In John J. Marciniak (ed.): *Encyclopedia of Software Engineering*, 2nd edition. New York: John Wiley.

[8] Mäkinen, E., T. Systä (2001). MAS – An Interactive Synthesizer to Support Behavioral Modeling in UML. *Proceedings 23th International Conference on Software Engineering.* 15-24.

[9] Nuseibeh, B., J. Kramer, A. Finkelstein (1994). A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering* **20**, 10. 760-773.

[10] Nuseibeh, B., S. Easterbrook, A. Russo (2000). Leveraging Inconsistency in Software Development. *IEEE Computer* **33**, 4. 24-29.

[11] *OMG Unified Modeling Language Specification* http://www.omg.org

[12] OMG (2003). *UML 2.0 Superstructure Specification.* OMG Final Adopted Specification, document ptc/03-08-02. http://www.omg.org/cgi-bin/doc?ptc/2003-08-02

[13] Whittle, J., J. Schumann (2000). Generating Statechart Designs from Scenarios. *Proceedings 22th International Conference on Software Engineering.* 314-323.

[14] Züllighoven, H. (1998). *Das objektorientierte Konstruktionshandbuch* (in German). Heidelberg: dpunkt Verlag [English version: *Object-Oriented Construction Handbook*. Morgan Kaufmann, 2004].