Approximation of (H)MSC semantics by Event Automata

Nikolai Mansurov[1)] **and Dmitri Vasura**

Institute for System Programming
25 B. Kommunisticheskaya
Moscow Russia
[1)]email: nick@ispras.ru

Abstract

This article describes a framework for requirements engineering with the so-called *use case scenario models*. Use case scenario models are based on *Message Sequence Charts (MSC) and High-level Message Sequence Charts (HMCS)*. In contrast to more conventional usage, (H)MSC scenarios are used to define behavior of a *single object* – the system under development for which requirements are modeled. Our formalization of use case scenario models facilitates requirements validation. Requirements validation is done by first automatically synthesizing executable model from scenarios and then performing *simulation* and *model checking* of the synthesized requirements model. The objective of our approach to requirements engineering is to have: 1) intuitive yet powerful formalism for modeling requirements; 2) intuitive structure of the synthesized executable models; 3) possibility to use powerful model checking and simulation tools. In order to achieve these objectives, instances of an (H)MSC model is independently mapped to the so-called *Event Automata.* The collection of Event Automata for all instances in the (H)MSC model provides some approximation of (H)MSC semantics. The goal of this paper is to formalize the nature of this approximation. We show that this approximation is adequate for the purpose of requirements modeling, and emphasize the benefits of the suggested approach: intuitive formal models of requirements, simple and flexible platform for generating code and using various powerful simulator and model checking tools, easy mapping of the analysis results back to the model. We outline the formalization of the use case models, event automata and synthesized requirements models. Additionally we describe a novel Video Cassette Recorder metaphor for visualization of use case scenario models, suitable for the problem domain experts, who are not familiar with formal notations. We also describe the Use Case Studio toolkit, in which these ideas are implemented.

Keywords: requirements engineering, formal models, semantics, verification, use cases, scenarios, message sequence charts, automatic synthesis of software, event automata, code generation, visualization

1. Introduction and background

Formal modeling techniques are often associated with high quality mission-critical software. However formal modeling can be also key to gaining time-to-market for development of "regular" software [1]. One of the challenges for wider adoption of formal modeling techniques in industry is to extend it to early development phases, starting with requirements capture and analysis [1,2]. This article describes a framework for requirements engineering based on the so-called *use case scenario models*.

Use case scenario models extend and formalize the well-known use case approach to software development [3]. Use case approach allows to specify behavior of the system without prematurely specifying its structure. The system is viewed as a "black box" and the structure of the environment of the system is considered instead. Behavior of the system (the set of functional requirements) is modeled as goal-directed interactions between the system and external actors, which constitute the environment of the system. The functionality of the system is partitioned into a number of use cases (typical, important services to external actors). Each use case is described by one or more *scenarios*.

Formalization of use case scenario model facilitates analysis and validation of requirements. Requirements validation is done by first automatically synthesizing executable models from scenarios and then performing simulation and model checking of the synthesized models. The objective of our approach to requirements engineering is to

have: 1) intuitive yet powerful formalism for modeling requirements; 2) intuitive structure of the synthesized requirements models; 3) possibility to use powerful model checking and simulation tools. We believe, that intuitive structure of the synthesized models is particularly valuable, because it allows to easily understand synthesized models, easily interpret the results of model checking in terms of the requirements and potentially even to use the synthesized model as the foundation for subsequent design phases.

Our formalization of use case scenario models is based on Message Sequence Charts (MSC) and High level Message Sequence Charts (HMSC) [4]. Combination of MSCs and HMSCs defines behavior of actors (external actors and the system actor) by specifying externally observable interactions between them. In particular, we want to specify the behavior of the system actor such that it produces an adequate response to various behaviors of external actors. As a consequence, our approach to scenarios is *not symmetrical*, since we are only interested in the behavior of one instance (corresponding to the system actor) with respect to all possible behaviors of other instances (corresponding to external actors). This lack of symmetry in treating scenarios is typical for a use case approach [3]. Scenarios are used to define behavior of a *single object* – the system under development for which requirements are modeled. This is in contrast to more conventional usages of scenarios (and the (H)MSC language) which usually describe collective behavior of distributed systems.

In order to achieve the above goals we introduced the concept of *Event Automata*. Event automaton models a single instance in an (H)MSC model. As will be demonstrated in this paper, independent synthesis of one event automaton from each instance of the (H)MSC model leads to simple and understandable executable models. The synthesis algorithm is simple and straightforward. Therefore, the concept of event automata is a convenient intermediate representation for subsequent generation of executable code within a certain target environment.

However, synthesized models based on event automata is only *an approximation* of the semantics of the input (H)MSC model. The synthesized requirements model is the collection of event automata, corresponding to all instances in the (H)MSC model, where each event automaton describes the behavior of a separate instance. The collective behavior of event automata of the synthesized model might be richer than the behavior, specified in the input (H)MSC model. Added behavior can include deadlocks, resulting from the lack of coordination between the instances.

Within the bigger context of the requirements engineering process, the added behavior resulting from inexact capture of (H)MSC semantics is likely to indicate poor understanding of the requirements. For example, a deadlock resulting from *non-local choice* [15] in the input (H)MSC model indicates a problem in the requirements model itself. Scenarios represent interactions between a system (as a black box) and external actors, therefore coordination between them can not be assumed. A deadlock indicates either a missing actor, which is responsible for coordination, or a missing interaction between existing actors and the system, or a missing requirement for the system being modeled. Early discovery of the need for additional coordination is valuable.

This idea is the central motivation of our approach to requirements engineering. Deadlocks in the synthesized model can be easily discovered by model checking tools such as SDT Validator tool [5] or SPIN [6]. Other types of added behavior can be discovered during the simulation of the synthesized model. Simplicity of the synthesized models allows to map discovered failures back to the initial requirements. The suggested requirements validation process is highly iterative, because each failed scenario can be revised and the updated "correct" scenario can be included to the set of requirements, from which a more precise model can be further synthesized.

Using (H)MSC scenarios to specify use cases allows designers to concentrate on *typical successful interactions*. On the other hand, subsequent event automata based synthesis and simulation *automatically discovers gaps in requirements* by generating

failures. Interesting enough, failures are exactly the result of the approximation of (H)MSC semantics by Event Automata.

Thus event automata-based synthesis of executable models from scenarios is a foundation for efficient CASE tools, extending the use of formal models to very early phases of the development process. The benefits of the suggested approach include intuitive formal models of requirements, simple and flexible platform for generating code and using various powerful simulator and model checking tools, easy mapping of the analysis results back to the model.

The paper has the following organization. Section 2 gives an outline of our formalization of use case scenario models. Section 3 introduces the concept of event automata, event automata based requirements models, their semantics and construction. Section 4 is central to our presentation. We describe the nature of the approximation of (H)MSC semantics by Event Automata based models. Section 5 contains a simple example, illustrating failures, which arise from the approximation of (H)MSCs by event automata. We show that the synthesized model has an unexpected added behavior resulting in a deadlock. We show that this deadlock can be discovered by SDT Validator tool. Section 8 provides some comparison to related research and concludes the presentation. Sections 6 and 7 outline a novel Video Cassette Recorder metaphor for visualization of use case scenario models and describe the UseCase Studio toolkit, which implements the above ideas.

2. Formalization of Use Case Scenario Models

2.1. Overview

The following sections outline formal definition of our requirements modeling language - the use case scenario models. Specification is written in ISO standard VDM-SL language [7]. Specification consists of a collection of type definitions. Due to the space limitations, all invariants (well-formedness conditions) as well as some trivial types and few auxiliary functions are not included. The structure of the VDM-SL specification of use case scenario models is presented at Figure 1 (as a UML class diagram [8]).

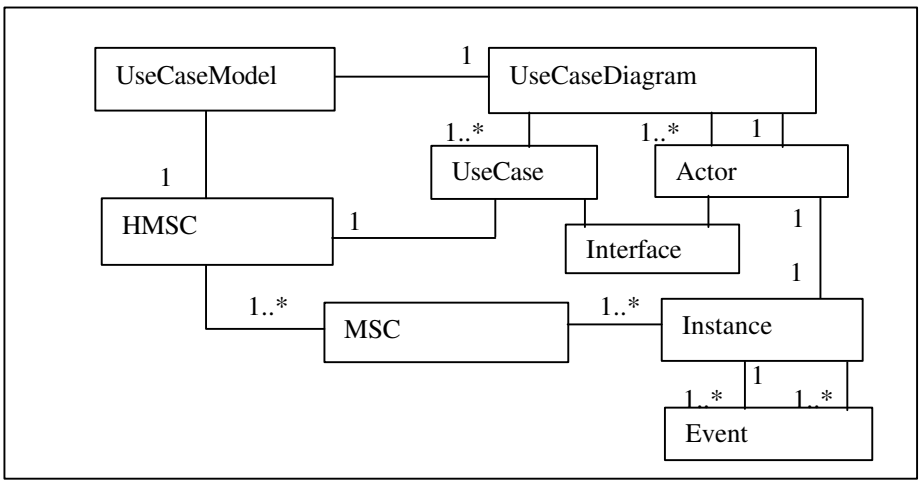

Figure 1. Overview of the formalization of Use Case Scenario models

The main type in our specification is *UseCaseModel*. As shown at Figure 1, UseCaseModel consists of a single *UseCaseDiagram* and a single *HMSC,* which references several MSCs. Each UseCaseDiagram contains multiple *Actors* and multiple *UseCases*. Each UseCase is formalized by another HMSC, which references several

MSCs. MSCs contains multiple *Instances*, which correspond to Actors, involved in the given UseCase. The system under developement (as the union of all use cases) is treated as another Actor (the so-called System Actor). Instances perform multiple *Events*, which also specify interactions between Instances. Actors interact with use cases through *Interfaces*. The idea of an *Interface* is elaborated in Section 6, where it plays the key role in visualization of use case scenario models according to the Video Cassette Recorder metaphor.

2.2. Message Sequence Charts

We start our formalization of use case requirements models with a subset of Message Sequence Charts [4]. A message sequence chart represents an *activity*, which is a sequence of interactions between instances to effect some desired operation or result. Our formalization does not consider timers, coregions, in-line expressions, references, gates, etc. [4].

In our formalization an *MSC* is a set of instances, involved in a certain activity. An *Instance* is a sequence of *Events.* All Events are assumed to be unique. If an Instance is created during the diagram, its first event is *StartEvent*. If an Instance is destroyed during the diagram, its last event is *TerminateEvent*. *Interaction* specifies communication between two Instances in the form of sending messages or creating one instance by another. Interaction binds related events of two different Instances. Sending a message between two instances involves a *SendEvent* in the sender Instance and a *ReceiveEvent* in the receiver Instance. Creating an instance involves a *CreateEvent* in the creator instance and a *StartEvent* in the newly created instance.

```
types
StartEvent ::  ;
CreateEvent ::
      name : InstanceName
      ;
SendEvent ::
      name : InstanceName
      message : MessageName
      ;
ReceiveEvent ::
      name : InstanceName
      message : MessageName
      ;
TerminateEvent ::  ;
Event = StartEvent | TerminateEvent |
      SendEvent | ReceiveEvent | CreateEvent ;


MSCevent ::
      id : EventId
      event : Event ;
Instance ::
      name : InstanceName
      events : seq of MSCevent
      ;
Interaction = map EventId to EventId;
MSC ::
      name : MSCname
      instances : set of Instance
      interaction : Interaction
      ;
```

2.3. High-Level MSCs

In our use case requirements models HMSCs are used as roadmaps to message sequence charts. An activity graph is a variation of a state machine in which states represent the performance of actions and the transactions are triggered by the completion of the actions. Actions refer to MSCs, which specify the activities in terms of interactions between instances. Our formalization does not consider conditions, node expressions and parallel frames [4].

HMSC is formalized as a set of HMSCnodes. Each *HMSCnode* represents a state. Transitions or flows connect states. HMSC has one *StartNode* and one *EndNode*. Every node in the graph must be reachable from the StartNode, i.e. the graph must be connected. We additionally assume, that the EndNode should be reachable from every node in the graph.

```
types
StartNode ::    ;
EndNode ::    ;
ReferenceNode ::
      ref : MSCname
      ;
ConnectionNode ::  ;
HMSCnode ::
      id : HMSCnodeId
      node : ReferenceNode | ConnectionNode | StartNode | EndNode
      ;
HMSC ::
      name : HMSCname
      nodes : set of HMSCnode
      flow : set of ( HMSCnodeId * HMSCnodeId)
      ;
```

2.4. Use Case Diagrams

Use case diagram shows *Actors* and *Use Cases* together with their relationships (the so-called *Interfaces*). Use Cases represent functionality of the system as observed by external users of the system [3,8]. Our Use case diagrams correspond to UML [8].

```
types
UseCase ::
      name : UseCaseName
      activity : HMSCname
      ;
Interface = ( Actor * UseCase );
UseCaseDiagram ::
            name : UseCaseDiagramName
            usecases : set of UseCase
            actors : set of Actor
            interfaces : set of Interface
            ;
```

2.5. Use Case Scenario Model

This section concludes the formalization of use case scenario models. *UseCaseModel* consists of a single *UseCaseDiagram* and a single *HMSC* (called `main`). This diagram describes composition of individual scenarios into complete behavior description. The UseCaseDiagram also contains all scenarios (formalized as MSCs) and

all intermediate HMSCs (or roadmaps), describing individual use cases. We assume, that each MSC contains the same set of instances, which corresponds to the set of all actors in the use case model.

```
types
UseCaseModel ::
      ucd : UseCaseDiagram
      main: HMSC
      scenarios : set of MSC
      roadmaps : set of HMSC
      ;
```

2.6. Semantics of Use Case Scenario Models

Definition 1. *Trace of events*. A trace of events, corresponding to an instance A in MSC M is a sequence S of events, such that S is equivalent to the complete sequence of events in the instance A in the MSC M. Two traces of events, which only differ in the origins of events received, are considered equivalent.

Definition 2. *Traversal of an HMSC.* Traversal of an HMSC H is a (finite) sequence T of MSC names, such that there exists a flow through H, starting at StartNode and finishing at the EndNode, which passes through the reference nodes with these MSC names (and through no other reference nodes) in the same order.

Definition 3. *Ground MSC.* Ground MSC corresponding to actor A in use case model U is an MSC, corresponding to some traversal of the HMSC main of U, in which all MSC names are substituted for the corresponding MSCs.

Definition 4. *Exemplary specified behavior.* Exemplary specified behavior of an actor A in use case model U is a trace of events S, corresponding to instance A in some ground MSC, corresponding to actor A in use case model U.

Definition 5. *Total specified behavior.* Total specified behavior of actor A in use case model U is a union of all exemplary specified behaviors S, corresponding to actor A in U.

Definition 6. *Semantics of the use case scenario model.* Semantics of a use case scenario model is a union of the set of total behaviors for all actors on the use case diagram, and the total behavior of the system actor.

3. Synthesized Requirements Models

This section describes the concept of Synthesized Requirements Models, which provides an executable approximation of use case scenario models.

3.1. Overview

The structure of the VDM-SL specification of synthesized requirements models is presented at Figure 2 (as a UML class diagram). The main type in this specification is *RequirementsModel*. RequirementsModel consists of several *Processes*. Each Process is represented by an EventAutomaton. *EventAutomaton* consists of *States*, which are connected by Transitions. Each *Transition* is labeled by an MSC *Event*. Some Events may refer to one or several *Messages*. Processes communicate by sending asynchronous

Messages. Each Process has a single *Input Port*, in which Messages are stored. Send Event in a sender process places a Message into the Input Port of the receiver process. Receive Event gets a Message from its own Input Port.

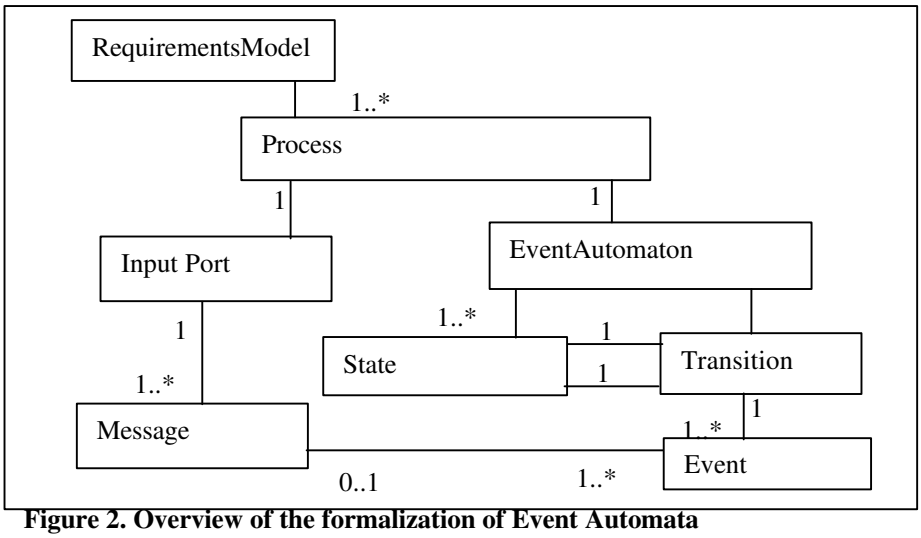

Figure 2. Overview of the formalization of Event Automata

3.2. Event Automata

The heart of the synthesized requirements model are the so-called Event Automata. The concept of event automata links traces of events with finite state automata from the theory of formal languages. An *Event Automaton* is a finite state automaton, such that its alphabet of input symbols is equivalent to the set of events of some MSC, and its input language is equivalent to valid event sequences, specified by that MSC. An Event Automaton is a *recognizer* for the language of event traces, corresponding to the total specified behavior of an individual instance. Therefore, Event Automaton does not make any distinction between send events and receive events. States in an Event Automaton correspond to the structure of the HMSC and sequences of events in each referenced MSC.

```
types
Transition = map (StateId * StateId) to ( Event | IdleEvent );

EventAutomaton ::
      name : InstanceName
      states : set of StateId
      transitions : Transition
      start_state : StateId
      end_state : StateId
```

3.3. Semantics of Event Automata

Algorithm 1: *Construct Event Automaton for a given Instance*:

1. construct initial states and transitions of the event automaton corresponding to nodes and flow of the HMSC. Transitions are labeled with IdleEvent. Set the start and end states of the event automaton as states, corresponding to the StartNode and EndNode of the HMSC, respectively.

2. for each MSC, create an Event Automaton corresponding to the sequence of events involving the given Instance
3. replace each state, corresponding to an ReferenceNode by the Event Automaton, corresponding to the referenced MSC

Definition 7. *Exemplary accepted behavior*. Exemplary accepted behavior of an event automaton E is a sequence M of MSC events, such that there exists a sequence of transitions through E, starting from the start_state and finishing in the end_state, where each transition is labeled by event from M, in the same order. Exemplary accepted behavior is a "sentence" (in the MSC event alphabet), "accepted" by automaton E.

Definition 8. *Total accepted behavior.* Total accepted behavior L of an event automaton E is a union of all exemplary accepted behaviors of E. L is the "language" (in MSC events alphabet), "accepted" by automaton E. Total accepted behavior defines semantics of the event automaton.

Proposition 1. Total accepted behavior of an event automaton E, constructed for instance A (of some use case scenario model U) is equivalent to the total specified behavior of this instance (with respect to U).

3.3. Requirements model

RequirementsModel type represents the top level of a use case scenario model. It captures "collective" behavior of several event automata. RequirementsModel is based on a well-known model of concurrent processes, communicating by asynchronous messages [9].

```
types
Process ::
      name : ProcessName
      input_port : seq of MessageName;
      graph : EventAutomaton
      ;
RequirementsModel ::
      name : RequirementsModelName
      processes : set of Process
      messages : set of MessageName ;
```

3.4. Semantics of Requirements Models

Algorithm 2: *Construct RequirementsModel for a given UseCaseModel*:

1. For each actor A in the UseCaseDiagram create an process PA, such that the graph of the Process PA is an EventAutomaton E for the instance A, corresponding to the current Actor A
2. Create Process PS for the system actor, such that the graph of this process is an EventAutomaton ES for the instance, corresponding to the system actor

In the context of the requirements model, event automata are interpreted as *generators* of the corresponding behaviors. In addition to automata semantics, side-effects of send and receive events on input ports are considered. Send event appends the message to the input port of the receiver process. Receive events depend on the state of the input port of the its host process.

Formalization of communicating state machines is done is a well-known way: the state of the requirements model is defined as a tuple, consisting of the state of each event automaton, and the state of the *input port* for each process (the sequence of messages, sent by other processes). Transition between states is performed in such a way, that one of the event automata performs one of the actions in its current state. ReceiveEvent can only be performed when the corresponding message is present in the input port. Any other event is performed immediately. SendEvent results in appending the corresponding message to the input port of the receiver process, similar to [9]. When no events can be performed by some event automaton in its current state, this event automaton is *blocked*. Semantics of the requirements model is defined by its state space and transition function.

As can be concluded from the above, requirements models are *executable*, i.e. a RequirementsModel can be simulated, or an equivalent specification in SDL [9] or Promela [6], etc. can be easily generated from it. Of course, the states and transitions of the event automaton need to be rearranged during generation, because the target state machine will distinguish between input states and other events.

4. The nature of approximating (H)MSCs by Event Automata

In this section we analyze the differences between (H)MSC models and event automata-based executable requirements models.

4.1. Approximation of "instance-based" (H)MSC semantics by event automata

Semantics of Use Case Scenario models, as defined in Section 2.6 already provides a certain approximation of (H)MSC semantics [4]. Precise semantics of (H)MSC models can be defined in terms of partial orders on the set of events on all instances, as defined by our ground MSCs, corresponding to the top-level HMSC of the use case model. An event trace (*involving all instances*!), i.e. a total order of events, is an exemplary specified behavior of an (H)MSC model, if it satisfies the partial order, defined by ground MSCs. However, we are interested in a non symmetric validation of requirements models, when the specified behavior of a selected instance is analyzed against different specified behaviors of all other instances. Therefore, we will first consider our "instance-oriented" semantics and analyze how it is approximated by Event Automata. Then we will discuss how Event Automata approximate precise semantics of (H)MSCs.

Definition 9. *Exemplary implemented behavior*. Exemplary implemented behavior of an actor A in requirements model R is a sequence S of MSC events, performed by the process P, corresponding to actor A in R such that there exists a sequence of transitions through the event automaton E(P), starting from the start_state and finishing in the end_state, where each transition is labeled by event from S, in the same order.

Definition 10. *Total implemented behavior*. Total implemented behavior of an actor A in requirements model E is a union of all exemplary implemented behaviors for A in E.

In contrast to accepted behavior, defined by an event automaton, implemented behavior depends on the behavior of other processes in E. In particular, ReceiveEvent with message M can be part of some exemplary implemented behavior, if another process performs the corresponding SendEvent. This leads us to the folowing important definition.

Definition 11. *Exemplary failed behavior.* Exemplary failed behavior of an actor A implemented in requirements model E is a sequence F of MSC events, performed by the process P, corresponding to actor A in R such that there exists a sequence of transitions through the event automaton E(P), starting from the start_state to some intermediate node N (different from the end_node), where each transition is labeled by event from S, in the same order, and process P is not able to perform any further transitions from node N.

Particular exemplary failed behavior F of an actor A in requirements model E is an equivalence class of paths through the state space of the requirements model E, such that A has performed sequence of events F, and other processes of E will *never* allow process A to make any further transition. This equivalence class will include the following paths through the state space of E:

- process A has performed sequence F, all other processes in E have reached their end states without sending any of the messages, which are responsible for the progress of A
- process A has performed sequence F, the set of processes, which can send messages, responsible for the progress of A, exercise some failed behavior themselves, other processes in E have reached their end state (deadlock).

Definition 12. *Total failed behavior.* Total failed behavior of an actor A in the requirements model E is the union of all exemplary failed behaviors of A in E.

Proposition 2. Total implemented behavior is equivalent to the union of total specified behavior and total failed behavior.

The proof of this proposition can be based on proposition 1 and the following three lemmas:

1) Total accepted behavior is included into total implemented behavior
2) There is no behavior other than accepted or failed
3) Failed behavior may exist

Proposition 2 is important. It means, that when we interpret scenarios in instance-based (non symmetric) way, we can synthesize a simple executable model based on event automata, which will generate failed behaviors. These failed behaviors are likely to indicate the gaps in our understanding of the requirements. On the other hand, the failed behaviors can be detected by model checking tools. The simplicity of the synthesis process allows to easily map failed behaviors back to the use case scenario model. In this context, the requirements engineer is concentrating on the well-understood success scenarios, and the model checking tool automatically discovers all failure scenarios, not addressed in the requirements model, and therefore ensures the completeness of the requirements the system actor with respect to all behaviors of the environment of the system.

4.2. Approximation of "complete" (H)MSC semantics by Event Automata

As was mentioned before, the union of all total specified behaviors of each instance in an (H)MSC model is already an approximation of (H)MSC semantics [4]. Unfortunately, when scenarios are interpreted in a symmetric way (and therefore, events from all instances are considered at the same time), our proposition 3 does not hold. The relation between the "total collective implemented behavior" and "total collective specified behavior" is more complex. There exist counterexamples of :

1) specified behavior, which is not implemented (at least while keeping the synthesized model simple). Observe that according to Definition 1, event automata are not sensitive to certain cases of *message overtaking*, which is a property of a "collective" behavior.
2) behavior which is implemented, but not specified. Event automata can implement more fine-grained interleaving of referenced MSCs, than allowed according to Definition 4.

5. Example of requirements engineering, supported by event automata

In this section we illustrate the suggested requirements engineering process, based on use case scenario models and proposition 2. First we consider a use case scenario model. Then we will show event automata, corresponding to the instances of the model. Then we will show an executable SDL model corresponding to the event automata-based requirements model. Then we will discuss the behavior of the synthesized model.

5.1. Use Case Scenario Model

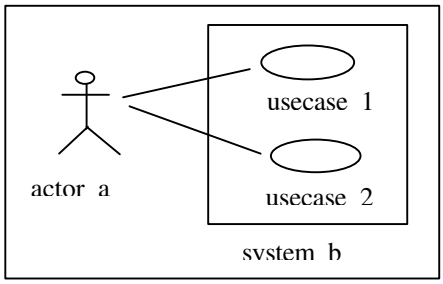

Figure 3. Example Use Case Diagram

Our example project captures requirements for a system called `system_b`. The environment for this system contains a single actor, called `actor_a`. Two use cases where identified: `usecase_1` and `usecase_2` (see use case diagram at Figure 3). Formalization of use cases by MSCs is illustrated at Figure 4. Figure 5 shows specification of the composition of MSCs into a complete behavior specification of the system actor `system_b`.

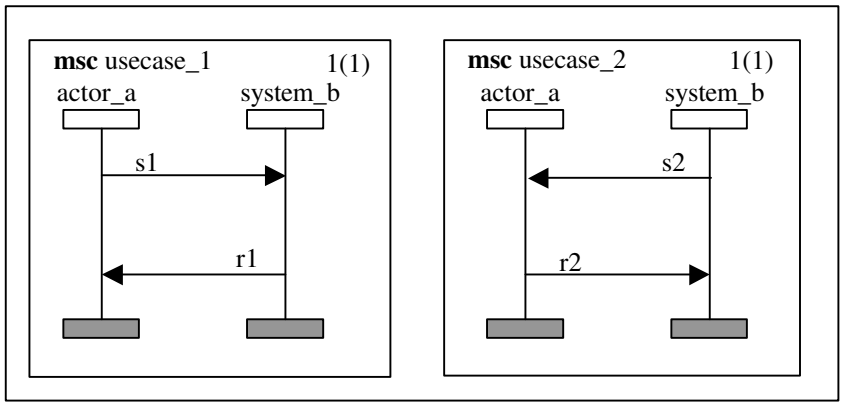

Figure 4. Example use cases, formalized by MSCs

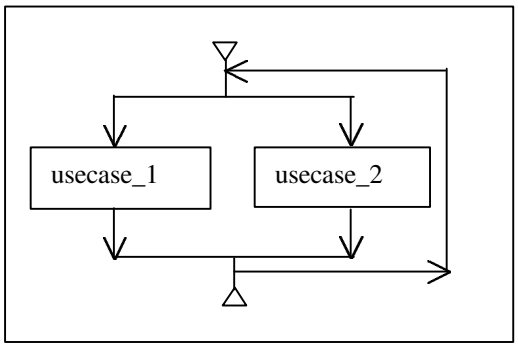

Figure 5. Example HMSC roadmap

5.2. Event Automata

This section illustrates event automata, corresponding to actors in the use case scenario model, introduced in the previous section. Event automata are illustrated at Figure 6.

Transitions in event automata are labeled with MSC events. Event automata do not differentiate between e.g. ReceiveEvent and SendEvent. For example, the start state of event automaton for `actor_a` has two outgoing transitions, one labeled with SendEvent, and another one - with ReceiveEvent.

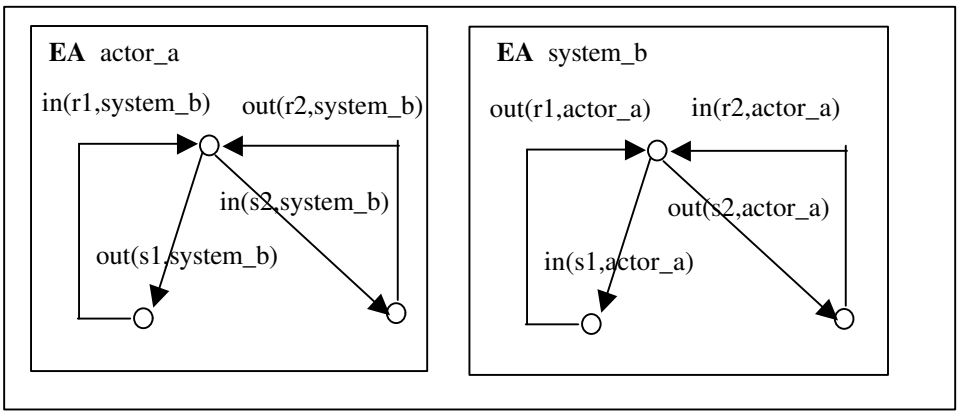

Figure 7. Event Automata

5.3. Generated SDL requirements model

This section illustrates SDL model, generated from the requirements model, introduced in section 5.1. Figure 7 illustrates SDL process definition, corresponding to the system actor `system_b`. Some states of an event automaton are mapped to SDL states, while some other states are mapped to SDL connectors (free actions). In particular, the start state of the event automaton is mapped onto an SDL free action `START_3`, which makes a non-deterministic choice between a transition, containing an `OUTPUT` statement, or a transition into an SDL state `ST_0`, where only INPUT statements can be performed. This illustrates the concept of a failed implemented behavior (definition 11): process `system_b` will first make a choice to proceed with an input statement (input event), but

the actual implementation (occurrence) of this event depends on the environment of this process (in our example - process `actor_a`).

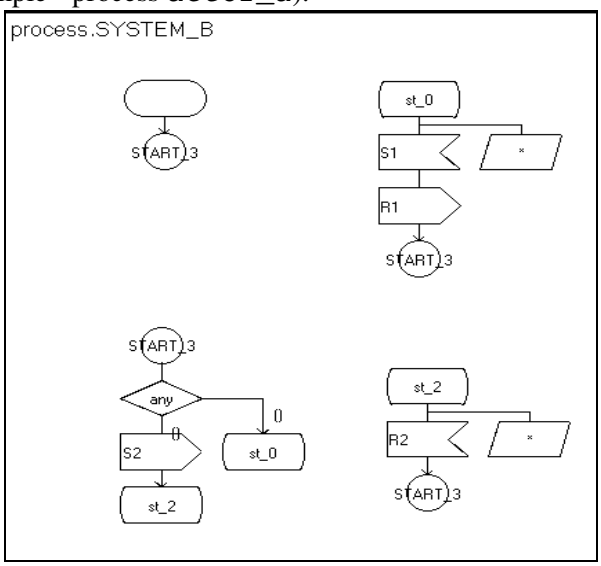

Figure 8. SDL process for system_b

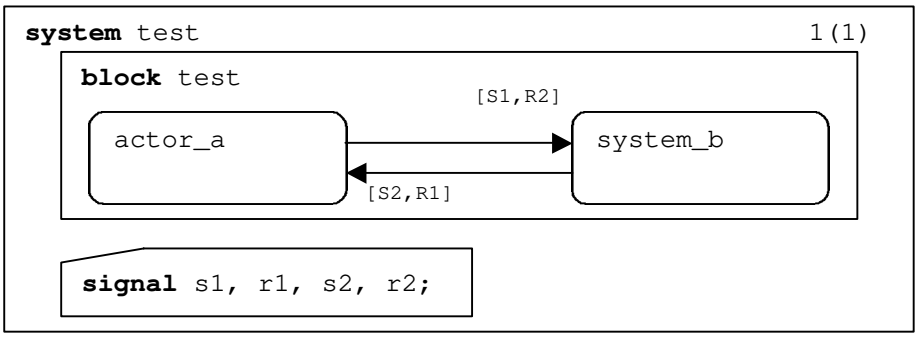

Figure 9. SDL block diagram for the requirements model

5.4. Implemented Behavior

The implemented behavior of the synthesized requirements model (which can be explored by simulating the generated SDL model) reproduces event traces from Figure 4. The synthesized model can also produce an infinite number of longer sequences of events, described by the HMSC at Figure 5. Additionally, the synthesized requirements model produces two classes of failed behavior, illustrated at Figure 9. In order to better illustrate the nature of the deadlock_2, we used the concept of a *found message* [4] to show, that each instance is waiting for a certain message.

Obviously, the two deadlocks indicate the lack of coordination between the system and external actor. Coordination can be achieved by e.g. introducing a timer, or additional messages. In the context of requirements engineering this situation indicates missing requirements.

Both situations can be automatically discovered by model checking tools using the generated model. Figure 10 illustrates the how SDT Validator tool [5] has presented both problems to the attention of a requirements engineer.

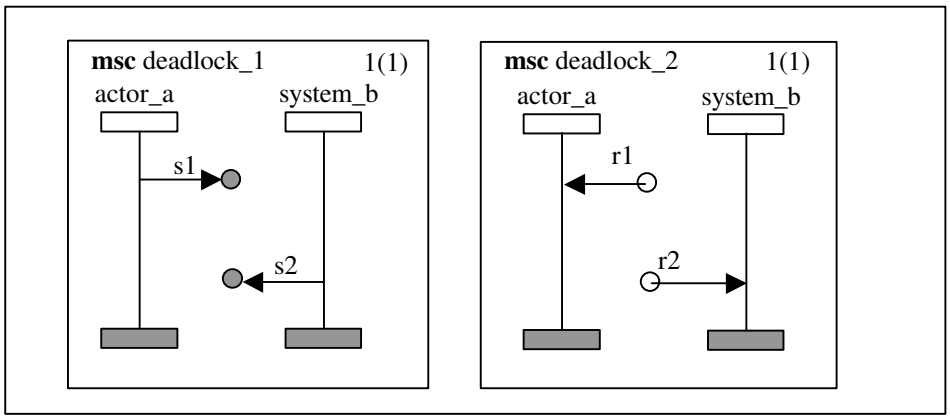

Figure 10. Failed behaviors, implemented by the requirements model

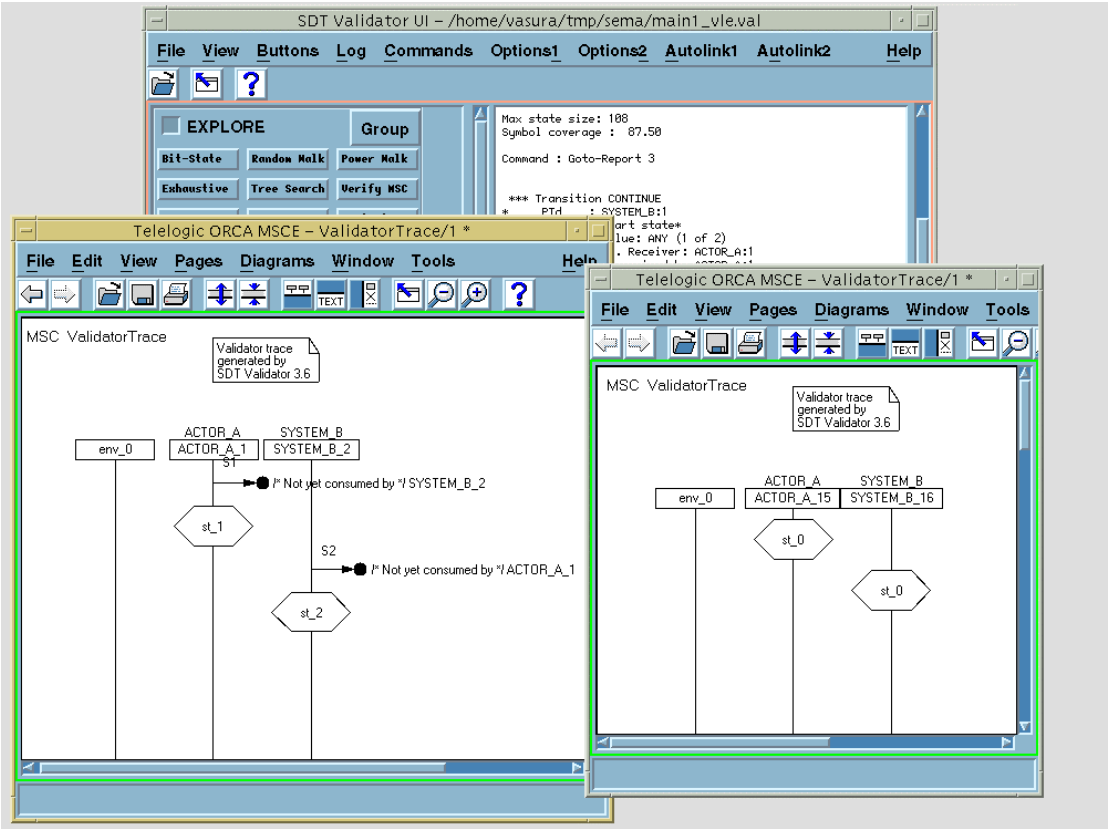

Figure 11. Failed behaviors discovered by SDT Validator

6. Visualization of use case scenario models

In this section we describe a novel approach to visualization of use case scenario models. We present a non-technical, user-friendly interface to create, simulate, validate use case scenario model, and update them based on the feedback from model checking tools. Our approach facilitates involvement of customers and other non-technical stakeholders into creating and validating formal requirements models.

We noticed some powerful similarities between engineering a use-case model and the film industry. Drawing upon some of these similarities allowed us to reconsider the way formal use case scenario model can be captured and visualized. We believe, that building the interface of a requirements engineering CASE tool upon the "film industry" terminology can make it significantly more familiar to non-technical audience.

use case model term	**Film industry term**
Formal use case model	Movie
Use case diagram	List of actors, cast
Actor	Actor
Role	Role
Use case	Episode
Scenario	Episode
Message Sequence Chart	Episode
HMSC	Collection of episodes
User Interface	Stage

Table 1. Similarities between film industry terms and formal requirements model terms

All similarities are quite well known in use case textbooks [9], except the last one. In film industry all events *occur* within the stage environment, while in the domain of engineering formal scenario-based models, some algebraic notation is used to represent sequences of events. The essence of suggested approach is to let events occur on the stage of the prototype user interface. This new approach allows to create episodes by *direct manipulation* with the elements of the user interface. The formal notation for event sequences can then be generated on the fly. On the other hand, an episode can be visualized by replaying it against the prototype user interface. With a suitable manipulation of graphical images, *illusion of direct manipulation* of the elements of the user interface can be created.

In order to achieve this, the prototype user interface has to be explicitly created before the episode is captured. Note, that this is recommended by UML methodology [8], however this was never enforced before.

In order to achieve capturing episodes by manipulation of the user interface, and replaying episodes by activating the user interface, the elements of the user interface need to be associated with events. Elements of the user interface (UI) are represented as state machines. States of the UI elements are associated with images. Transitions between states are associated with events. Additionally, transitions can be associated with animations for replaying episodes.

We suggest the Video Cassette Recorder (VCR) metaphor for engineering use case scenario models. The graphical user interface to individual episodes can be similar to a VCR panel with “record”, “playback”, “fast forward”, “reverse”, and “stop” buttons.

There exists another powerful similarity to film industry: assembling episodes into HMSCs is similar to montage of episodes into film. This leads to "Montage Table" interface to arranging episodes into more complete behaviour descriptions where individual episodes can be selected, composed together into higher level scripts (or episode flows) and stored. There is an important difference between movie industry and use case modelling: in film industry, a film is strictly a linear composition of episodes, i.e. an episode is a short sequence of events, while a film (as a linear composition of episodes) is a larger sequence of events. However, in use case scenario models, the composition of scenarios into a use case is non-linear (i.e. a use case can include alternatives, repetitions of scenarios, etc.). Therefore, a use case is a different object, than a linear sequence of events. However, the similarity is restored, if we consider collections of episodes (existing in the creative laboratory of a film-maker).

The principles of a user-friendly tool for engineering use-case models can be summarised as follows:

- prototype user interface is created before use cases are captured;
- use case is captured as the result of direct manipulation of the elements of the user interface (sequences of events are generated as the result of performing sequences of activation's of the user interface elements);
- use case is re-played against the user interface (sequence of events causes the activation of the corresponding elements of the user interface);
- problematic sequences, discovered by the model checking tools are re-played against the user interface;
- The key "metaphor" of the visualization of use case scenario models is a Video Cassette Recorder and the corresponding metaphor of a Montage Table;
- all formal notations can be hidden from the requirements engineer, including (H)MSC for specifying scenarios, event automata for simulating use cases, SDL or any other target execution platform, and model checking tools for validation of use case scenario models; however, formal notations are available and can be used by advanced users (same as shortcuts in graphical tools).

7. UseCase Studio toolkit

`

UseCase Studio is a model-based requirements engineering toolkit, which provides complete support for requirements capture and analysis with use case scenario models. The UseCase Studio implements the Video Cassette Recorder metaphor for use case capture and replay. The UseCase Studio also implements the above described event automata-based simulation and validation of use case scenario models.

The UseCase Studio consists of the synthesizer kernel [12] which implements analysis of use case scenario models and synthesizes event automata, and the set of the following visualization tools:

Use Case Editor	Visual editor for UML Use Case Diagrams
Interface Editor	Visual editor (composer) for formalized user interfaces, with support for providing associations with events
Scenario Recorder	Visual tool with a VCR-like interface to capture sequences of events by direct manipulation with the user interface elements of the formalized user interface; also for replaying stored scenarios against user interfaces
Episode Editor	Visual Editor for HMSCs, acting as roadmaps for sequence diagrams
Episode Simulator	Visual tool for simulating HMSCs
MSC Editor	Visual Editor for Message Sequence Charts or UML Sequence Diagrams
Model Navigator	Visual information tool, which provides interface to all artifacts of the use case scenario model

7.1. Interface Editor

Graphical user interfaces are associated with links between use cases and external actors on use case diagrams (Section 2.4). **Interface Editor** of the UseCase Studio allows

to prototype a GUI visually and intuitively, without having any experience in programming such interfaces for industry applications.

The editor looks like a usual graphical editor. It has a palette of user interface elements. UI elements include buttons, text labels, text edit controls, check buttons, list controls and user defined images. UI elements can be visually arranged on the screen to create the desired layout of user interface.

Active UI elements (elements of the user interface, which is control the behaviour of the system, e.g. button, check button and elements of list control), can be associated with a sequence of *active events.* Events will be generated in the order specified in control properties dialog, whenever the input UI element is activated by the user.

Passive UI elements (elements of the user interface, which reflect the current state of the system e.g. check button, image, list control and text label), can be associated with a sequence of *input events*. These events are further associated with different *states* of this UI element (e.g. on/off state for check button, different images for image pattern, different text in text label and positions of list control). Changes of the state of the passive UI elements are caused by active UI elements.

Prototype user interfaces are used to capture scenarios, as well as to present traces for validation. Association of UI elements with MSC events has dual use: in "record" mode this association is used to create (edit) a sequence diagram; in "playback" mode this association is used to visualize a sequence diagram.

7.2. Scenario Recorder

When all interfaces to actors which deals with certain use case and system actor interface are specified, scenarios for this use case can be recorded using **Scenario Recorder Tool** of the UseCase Studio.

To start recording one should select the use case in the Navigator window and Scenario Recorder Tools from Navigator tool bar or menu. Scenario Recorder and MSC Editor are started and interfaces for all external actors and system actor interface are displayed. New MSC Diagram is created in MSC Editor with empty instance axes for the external and system actors.

Now it is possible to record sequences of events, corresponding to the behavior of external actors and the system by directly activating (i.e. pressing, selecting, switching etc.) UI elements of user interfaces. Sequences of events activated through UI elements will simultaneously appear in MSC Editor.

When a UI element is activated by the user, the sequence of events, associated with this UI element is generated (i.e. inserted into the MSC Diagram for the corresponding actor in the order specified). If some UI elements in this or other user interfaces contain any of these events in their inactive events set, their state is changed accordingly.

Scenario Recorder panel looks like a VCR control panel. Using "back" and "fast back" buttons from the panel user may back for some steps and then start recording again and replay the former sequence of actions using buttons "forward" and "fast forward". Back and forward operations are reflected on the MSC Diagram as well as on dialog boxes, so user does not need to make any manual changes in the MSC Editor.

During simulation all events traversed by the Scenario Recorder are visualized against User Interfaces as changing states of UI elements. In contrast to the "record" mode, active events assigned to controls are considered to be input and thus influence the state, what leads to such effects, as buttons being pressed "by themselves" when input and active event assigned to this button is executed by the Scenario Recorder. This creates an illusion of direct manipulation of the UI elements.

7.3. Episode Editor and Episode Simulator

When all scenarios for a certain use case are recorded in Scenario Recorder, the **Episode Editor** can be used to combine these scenarios into an HMSC Diagram, which specifies complete behaviour of the system in the given use case [6].

When the HMSC Diagram is completed, the use case can be simulated using the Episode Simulator tool of the UseCase Studio. User Interfaces for all actors, which use system in this use case, are started and Episode Simulator highlights the start node of the HMSC.

The Episode Simulator has a VCR-control panel. Step-by-step simulation of the HMSC can be performed using “forward”, “play” and “back” buttons on the Episode Simulator panel. When “play” button is pressed, simulator steps through the HMSC, i.e. makes a transition from the currently selected symbol to the next symbol by the connectivity line. If there are several possible transitions from the currently selected symbol, Episode Simulator highlights all possible symbols, to which the transition may be done, and asks the user to make a choice.

When an MSC reference symbol is reached, the corresponding MSC Diagram is loaded into the Sequence Diagram Editor and simulation is continued in the Scenario Recorder.

8. Conclusions and Related Work

We presented the framework for requirements engineering based on formal use case scenario models. The current article presented some results, not available in the previous works of our group at the Department for CASE tools of the Institute for System Programming [10,11,12]. This research is done within a broader context of developing an Accelerated Development Methodology (ADM), which uses the new generation formal-model based CASE tools to improve time-to-market for development of telecommunications software [1]. The objectives of our approach to requirements engineering are to have: 1) intuitive yet powerful formalism for modeling requirements; 2) intuitive structure of the synthesized requirements models; 3) possibility to use powerful model checking and simulation tools. Intuitive structure of the synthesized models is particularly attractive, because it allows to easily understand synthesized models, easily interpret the results of model checking in terms of the requirements and potentially even to use the synthesized model as the foundation for subsequent design phases.

The use case scenario models extends and formalizes the use case approach to software engineering [3,8]. Notations, like UML and (H)MSC are becoming more and more accepted, which addresses the "*accessibility barrier*" [1] for wider adoption of formal model based CASE tools into industry. Until recently, many mathematical based formal methods existed, but were not adopted into industry, because on an insufficient number of experts available for these notations [1].

Several research groups are using scenarios in the form of MSCs or UML Sequence Diagrams as a front-end for model checking tools [15,16,17,18]. The majority of these approaches use symmetrical approach to scenarios. Our non-symmetric approximation is more suitable for requirements engineering of "regular" (not necessarily distributed) software, and fits better with use case based approach.

Precise formal semantics of combined usage of HMSCs and MSCs is still a field of active research [13,14,17,18]. Particularly interesting approach [13] uses the formalism of graph grammars for capturing the "collective" semantics of (H)MSC.

Using (H)MSC scenarios to specify use cases allows designers to concentrate on *typical successful interactions*. On the other hand, subsequent event automata based

synthesis and simulation *automatically discovers gaps in requirements* by generating failures. Interesting enough, failures are exactly the result of the approximation of (H)MSC semantics by Event Automata.

Thus event automata-based synthesis of executable models from scenarios is a foundation for efficient CASE tools, extending the use of formal models to very early phases of the development process. The benefits of the suggested approach include intuitive formal models of requirements, simple and flexible platform for generating code and using various powerful simulator and model checking tools, easy mapping of the analysis results back to the model.

8. References

1. R. Probert, N. Mansurov, Improving time-to-market using SDL tools and techniques (tutorial), in Proc. 9th SDL Forum, Montreal, Canada, June 21-26, 1999
2. G. Holzmann, Early fault detection tools, in T. Margaria, B. Steffen (Eds), TACAS'96, LNCS, vol. 1055, pp. 1-13, Springer Verlag, 1996
3. I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard, Object-Oriented Software Engineerings: A Use Case Driven Approach, Addison-Wesley, Reading, MA, 1992
4. ITU-T, Rec. Z.120, Message Sequence Charts (MSC), Geneva, 1996
5. Telelogic AB, SDT Validator documentation, Malmoe, Sweden, 1999
6. G. Holzmann, Design and Validation of Computer Protocols, Prentice Hall, Englewood Cliffs, N.J. 1991
7. D.J. Andrews (ed). Information technology - Programming Languages, their environments and system software interfaces - Vienna Development Method - Specification Language - Part 1. Base Language. ISO, Geneva, December 1996. ISO/IEC 13817-1
8. G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1998
9. ITU-T, Rec. Z.100, Specifications and Description Language (SDL), 1996
10. N. Mansurov, D. Zhukov, Automatic Synthesis of SDL Models in Use Case Methodology, in Proc. SDL'99 The Next Millenium, ed. R. Dssouli et. at., Elsevier, NY, 1999, pp. 225-240
11. N. Mansurov, Automatic Synthesis of SDL from MSC in Forward and Reverse Engineering, in Proc. VFM'99, Ed. S. Mauw, et.al., Eindhoven, The Netherlands, 1999, Eindhoven University of Technology, report 99-08, pp. 44-64
12. D.Vasura, D.Zhukov, N.Mansurov, MOST-SDL tool documentation, http://case.ispras.ru, 1999
13. L. Helouet, C. Jard, B. Caillaud, An effective equivalence for sets of scenarios, represented by HMSCs, IRISA Technical report N. 1205, 1998
14. I. Krueger, R. Grosu, P. Scholz, M. Broy, From MSCs to Statecharts, in DIPES'98, Kluwer, 1999
15. H. Ben-Abdallah, S. Leue, Syntactic Detection of Process Divergence and non-local Choice in Message Sequence Charts, in E. Brinksma (Ed), Proc. TACAS'97, Enschede, The Netherlands, April 1997, LNCS, Vol. 1217, pp. 259-274, Springer Verlag, 1997
16. R. Alur, G. Holzmann, D. Peled, An analyzer for message sequence charts, in T. Margaria, B. Steffen (Eds), TACAS'96, LNCS, vol. 1055, pp. 35-48, Springer Verlag, 1996
17. W. Damm, D. Harel, LSCs: Breathing Life into Message Sequence Charts, Weizmann Institute of Technology Report CS98-09, April 1998
18. X. Li, J. Lilius, Timing Analysis of UML Sequence Diagrams, Turku Centre for Computer Science, TUCS Technical Report 281, May 1999