Documenting the Progress of the System Development*

Marta Pląska[1], Marina Waldén[1] and Colin Snook[2]

[1] Åbo Akademi University/TUCS, Joukahaisenkatu 3-5A, 20520 Turku, Finland
[2] University of Southampton, Southampton, SO17 1BJ, UK

Abstract. While UML gives an intuitive image of the system, formal methods provide the proof of its correctness. We can benefit from both aspects by combining UML and formal methods. Even for the combined method we need consistent and compact description of the changes made during the system development. In the development process certain design patterns can be applied. In this paper we introduce progress diagrams to document the design decisions and detailing of the system in successive refinement steps. A case study illustrates the use of the progress diagrams.

Keywords: Progress diagram, Statemachines, Stepwise development, Refinement, UML, Event-B, Action Systems, Graphical representation.

1. Introduction

For complex systems the stepwise development approach of formal methods is beneficial, especially considering issues of ensuring the correctness of the system. However, formal methods are often difficult for industrial practitioners to use. Therefore, they need to be supported by a more approachable platform. The Unified Modelling Language (UML) is commonly used within the computer industry but, currently, mature formal proof tools are not available. Hence, we use formal methods in combination with the semi-formal UML.

For a formal top-down approach we use the Event B formalism [10] and associated proof tool to develop the system and prove its correctness. Event-B is based on Action Systems [4] as well as the B Method [1], and is related to B Action Systems [17]. With the Event-B formalism we have tool support for proving the correctness of the development. In order to translate UML models into Event B, the UML-B tool [14] is used. UML-B is a specialisation of UML that defines a formal modelling notation combining UML and B.

The first phase of the design approach is to state the functional requirements of the system using natural language illustrated by various UML diagrams, such as statechart diagrams and sequence diagrams that depict the behaviour of the system. The system is built up gradually in small steps using superposition refinement [3, 9]. We rely on patterns in the refinement process, since these are the cornerstones for creating *reusable* and *robust* software [2, 7]. UML diagrams and corresponding Event B code are developed for each step simultaneously. To get a better overview of the design process, we introduce the *progress diagram*, which illustrates only the refinement-affected parts of the system and is based on statechart diagrams. Progress diagrams support the construction of large software systems in an incremental and layered fashion. Moreover, they help to master the

* Work done within the RODIN-project, IST-511599

complexity of the project and to reason about the properties of the system. We illustrate the use of the diagrams with a case study.

Design patterns in UML and B have been studied previously. Chan et al. [6] work on identifying patterns at the specification level, while we are interested in refinement patterns. The refinement approach on design patterns was presented by Ilič et al. [8]. They focused on using design patterns for integrating requirements into the system models via model transformation. This was done with strong support of the Model Driven Architecture methodology, which we do not consider in this paper. Instead we provide an overview of the development from the patterns.

The rest of the paper is organised as follows. In Section 2 we give an overview of our case study, Memento, from a general and functional perspective. An abstract specification is presented as a graphical, as well as a formal representation in Section 3. Section 4 describes stepwise refinement of the system and introduces the idea of progress diagrams. The system development is analysed and illustrated with the progress diagrams relying on the case study. We conclude with some general remarks in Section 5.

2. Case study – Memento application

The Memento application [13] that is used as a case study in this paper is a commercial application developed by Unforgiven.pl. It is an organiser and reminder system that has lately evolved into an internet-based application. Memento is designed to be a framework for running different modules that interact with each other.

In the distributed version of Memento every user of the application must have its own, unique identifier, and all communication is done via a central application server. In addition to its basic reminder and address book functions, Memento can be configured with other function modules, such as a simple chat module. Centralisation via the use of a server allows the application to store its data independently of the physical user location, which means that the user is able to use his own Memento data on any computer that has access to the network.

The design combines the web-based approach of internet communicators and an open architecture without the need for installation at client machines. During its start-up the client application attempts to *connect to a central server*. When the connection is established, the *preparation phase* begins. In this phase the user provides his/her unique identifier and password for authorisation. On successful login the server responds by sending the data for the account including a list of contacts, news, personal files etc. Subsequently the *application searches for modules* in a working folder and attempts to initialise them, so that the user is free to run any of them at any time. During execution of the application, *commands from the server and the user are processed* at once. Memento translates the requested actions of the user to internal commands and then handles them either locally or via the server. Upon a termination command Memento *finalises all the modules*, saves the needed data on the server, logs out the user and *closes the connection*. To minimise the risk of losing data, in case of fatal error, this termination procedure is also part of the fatal exception handling routine.

3. Abstract specification

3.1. UML-models

We use the Unified Modelling Language™ (UML) [5], as a way of modelling not only the application structure, behaviour, and architecture of a system, but also its data structure. UML can be used to overcome the barrier between the informal industry world and the formal one of the researchers. It provides a graphical interface and documentation for every stage of the (formal) development process. Although UML offers miscellaneous diagrams for different purposes, we focus on two types of these in our paper: sequence diagrams and statechart diagrams.

The sequence diagram can be used within the development of the system to show the interactions between objects and in which order these interactions occur. The diagram can be derived directly from the requirements. Furthermore, it can give information on the transitions of the statemachines. The interaction between entities in the sequence diagram can be mapped to self-transitions on the statechart diagram to model communication between the modelled entity and its external entities.

In our case study the external entities are the server and the users interacting with the modelled entity Memento. An example of a sequence diagram for the application is given in Fig. 1, where part of the requirements (the emphasized text in Section 2) concerning the server connection and the program preparation phase is shown. In the diagram we describe the initialisation phase of the system, which consists of establishing a connection (in the connection phase) and then preparing the program (in the preparation phase). The first of these actions requires the interaction with the server via an internet connection. The second action requires user interaction as well. The described interaction (in Fig. 1) is transferred to a statechart diagram as transition *tryInit* (to later be refined to the transitions *tryConn* and *tryPrep* as in section 4.1).

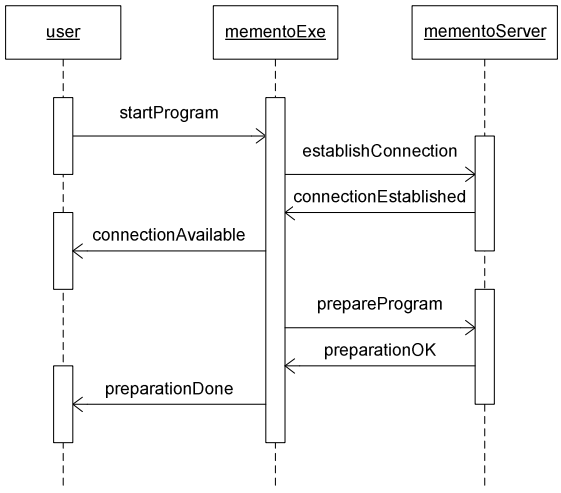

Fig. 1. Sequence diagram presenting the object interaction in the *initialisation phase*

In statechart diagrams objects consist of states and behaviours (transitions). The state of an object depends on the previous transition and its condition (guard). A statechart diagram provides graphical description of the transitions of an object from one state to another in response to events [12, 11]. The diagram can be used to illustrate the behaviour of instances of a model element. In other words, a statechart diagram shows the possible states of the object and the transitions between them.

The statemachine depicting the abstract behaviour of Memento is shown in Fig. 2. The first phase is to initialise the system by communicating with the server, which is modelled with the event *tryInit*. When initialisation has been successfully completed, the transition goReady brings the system to the state ready, where it awaits and processes the user and server commands. Upon the command *close*, the system enters the finalisation phase, which leads to the system cleanup and proper termination.

The detection of errors in each phase is taken into consideration. In the model, the errors are captured by transitions targeting the suspended state (*susp*), where error handling (rollback) takes place. The system may return to the state where the error was detected, if the error happens to be recoverable. If the error is non-recoverable, the fatal termination action is taken and the system operation finishes. Any error detected during or after finalisation phase is always non-recoverable.

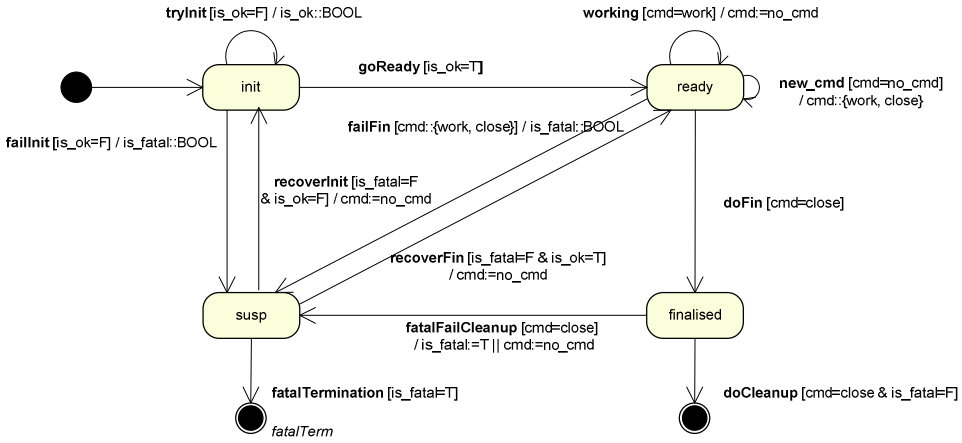

Fig. 2. The abstract statemachine of Memento

3.2. Formal specification

In order to be able to reason formally about the abstract specification, we translate it to the formal language Event B [10]. An Event-B specification consists of a model and its context that depict the dynamic and the static part of the specification, respectively. They are both identified by unique names. The context contains the sets and constants of the model with their properties and is accessed by the model through the SEES relationship [1]. The dynamic model, on the other hand, defines the state variables, as well as the operations on these. Types and properties of the variables are given in the invariant. All the variables are assigned an initial value according to the invariant. The operations on the variables are given as events of the form **WHEN** guard **THEN** substitution **END** in the Event-B specification. When the guard evaluates to true the event is said to be enabled. The events are considered

to be atomic, and hence, only their pre and post states are of interest. In order to be able to ensure the correctness of the system, the abstract model should be consistent and feasible [10].

Each transition of a statechart diagram is translated to an event in Event-B. Below we show the Event B-translation of the statemachine concerning the initialisation (state *init*) of the cooperation with the server in Fig. 2:

MODEL Memento
SEES Data
VARIABLES is_fatal, is_ok, cmd, state
INVARIANT is_fatal ∈ BOOL ∧ is_ok ∈ BOOL ∧ cmd ∈ CMD ∧ state ∈ STATE ∧
(state=init ⇒ cmd=no_cmd) ∧ ...
INITIALISATION is_fatal:=FALSE || cmd:=no_cmd || is_ok:=FALSE || state:=init
EVENTS
tryInit = **WHEN** state=init ∧ is_ok=FALSE **THEN** is_ok :∈ BOOL **END**;
failInit = **WHEN** state=init ∧ is_ok=FALSE **THEN** state:=susp || is_fatal :∈ BOOL **END**;
recoverInit= **WHEN** state=susp ∧ is_ok=FALSE ∧ is_fatal=FALSE **THEN** state:=init || cmd:=no_cmd **END**;
goReady = **WHEN** state=init ∧ is_ok=TRUE **THEN** state:=ready **END**;
...
END

The variables model a proper initialisation (*is_ok*), occurrence of a fatal error (*is_fatal*), as well as the command (*cmd*) and the state of the system (*state*). Initially no command is given and the initialisation phase is marked as not completed (*is_ok := FALSE*). The guards of the transitions in the statechart diagram in Fig. 2 are transformed to the guards of the events in the Event B model above, whereas the substitutions in the transitions are given as the substitutions of the events. The feasibility and the consistency of the specification is then proved using the Event-B prover tool.

4. Modelling refinement steps

It is convenient not to handle all the implementation issues at the same time, but to introduce details of the system to the specification in a stepwise manner. Stepwise refinement of a specification is supported by the Event-B formalism. In the refinement process an abstract specification *A* is transformed into a more concrete and deterministic system *C* that preserves the functionality of *A*. We use the superposition refinement technique [3, 9, 17], where we add new functionality, i.e., new variables and substitutions on these, to a specification in a way that preserves the old behaviour. The variables are added gradually to the specification with their conditions and properties. The computation concerning the new variables is introduced in the existing events by strengthening their guards and adding new substitutions on these variables. New events, assigning the new variables, may also be introduced.

System *C* is said to be a correct refinement of *A* if the following proof obligations are satisfied [10, 15, 17]:

1. The initialisation in *C* should be a refinement of the initialisation in *A*, and it should establish the invariant in *C*.
2. Each old event in *C* should refine an event in *A*, and preserve the invariant of *C*.
3. Each new event in *C* (that does not refine an event in *A*) should only concern the new variables, and preserve the invariant.
4. The new events in *C* should eventually all be disabled, if they are executed in isolation, so that one of the old events is executed (non-divergence).

5. Whenever an event in *A* is enabled, either the corresponding event in *C* or one of the new events in *C* should be enabled (strong relative deadlock freeness).

6. Whenever an error detection event (event leading to the state *susp)* in *A* is enabled, an error detection event in *C* should be enabled (partitioning an abstract representation of an error type into distinct concrete errors during the refinement process [16]).

The tool support provided by Event-B allows us to prove that the concrete specification *C* is a refinement of the abstract specification *A* according to the proof obligations (1) - (6) given above.

In order to guide the refinement process and make it more controllable, refinement patterns [11] can be used. The size of the system grows during the development making it difficult to get an overview of the refinement process. In this paper we introduce progress diagrams to give an abstraction and graphical-descriptive view documenting the applied patterns in each step.

4.1. Progress diagrams

We introduce the idea of progress diagram in the form of a table that is divided into a description part and a diagram part. With this type of table we can point out the design patterns derived from the most important features and changes done in the refinement step. It provides compact information about each refinement step, thereby indicating and documenting the progress of the development. The tabular part briefly describes the relevant features or design patterns of the system in the development step. Moreover, it depicts how states and transitions (initiated, refined or anticipated) are refined, as well as new variables that are added with respect to these features. The diagram part gives a supplementary view of the current refinement step and is in fact a fragment of the statechart diagram.

During the development we benefit from the progress diagram, as we concentrate only on the refined part of the system. The combination of descriptive and visual approaches to show the development of the system gives a compact overview of the part that is the current scope of development. This enables us to focus on the details we are most interested in, and provides a legible picture of the (possibly complex) systems development. The visualisation helps us to better understand the refinement steps and proofs that need to be performed. Progress diagrams do not involve any mathematical notation and are, therefore, useful for communicating the development steps to non-formal methods colleagues. We will illustrate the use of progress diagrams with our case study Memento.

Fig. 3 depicts the progress diagram of the *first refinement step*, where states are partitioned into substates and transitions are added with respect to these. Partitioning the state *init* indicates that the initialisation phase is divided into a connection (state *conn*) and a preparation (state *prep*) phase, that both need the cooperation with the server. The state *susp* is treated in a similar way. Namely, the hierarchical substates *sc*, *sp*, *sr* and *sf* are created, implying that there are in fact various ways of handling the errors, corresponding to the states *conn*, *prep*, *ready* and *finalised*. Thereby, more elaborate information about conditions of error occurrence is added. Note that introducing hierarchical substates corresponds not only to a more detailed model in the structural sense, but also in the functional sense. The transitions (events) *tryInit*, *failInit* and *recoverInit* are refined to more detailed ones taking into account the partitioning of the initialisation phase. The self-transition *tryInit* is refined by two events, *tryConn* and *tryPrep*, which remain self-

transitions for the states *conn* and *prep,* respectively. The error handling is refined by events: *failConn* and *recoverConn* for the substate *conn*, and *failPrep* and *recoverPrep* for the substate *prep*. The anticipating transition *cont* is added between the new substates *conn* and *prep*. The new variables are introduced to control the system execution flow. Note that for the substates *sr* and *sf* there are separate diagram parts.

Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
1st refinement step: • creating hierarchical substates (in states *int* and *susp*) • adding new transitions concerning the substates	*init*	*conn* *prep*	*tryInit*	*tryConn, tryPrep*	*is_conn* *is_prep* *wwaited*
			-	*Cont*	
	susp	*sc, sp,* *sr, sf*	*failInit*	*failConn, failPrep*	
			recoverInit	*recoverConn, recoverPrep*	

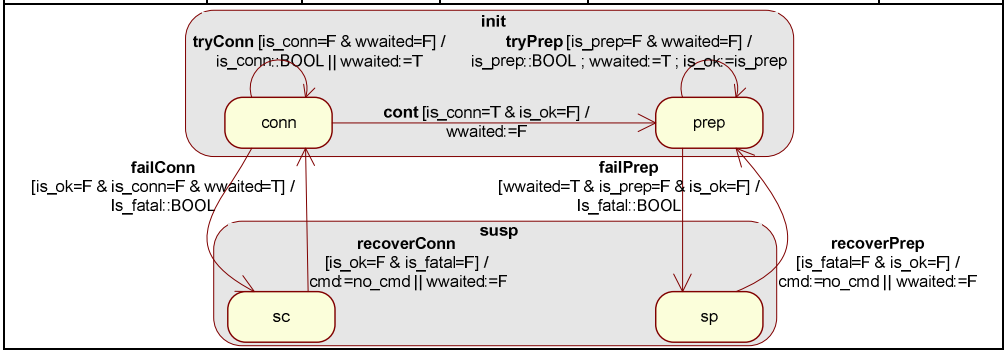

Fig. 3. Progress diagram of the first refinement step of Memento

As the refined specification is translated to Event B for proving its correctness, the progress diagram can provide an overview of the proof obligations needed for the refinement step concerning the refined and the anticipating events. In the progress diagram the refined events are the ones given in the column "Refined Transitions" that have a corresponding event in the column "Transitions" (Proof Obligation (2)). For example in Fig. 3 events *tryConn* and *tryPrep* refine *tryInit*. Also the anticipating events are given in the column "Refined Transitions" (event *cont* in Fig. 3). However, they do not have a corresponding event in the column "Transitions". They may only assign the variables in column "New Variables" according to the invariant (Proof Obligation (3)). Furthermore, the non-divergence of the anticipating transitions (Proof Obligation (4)) is indicated in the diagram part by the fact that these transitions do not form a loop [15]. From the columns "Transitions" and "Refined Transitions" also partitioning of the error detection events is indicated (Proof Obligation (6)). In Fig. 3 the error detection event *failinit* is partitioned into *failConn* and *failPrep*.

The result of the first refinement step is shown in the statechart diagram in Fig. 4. When comparing this diagram to the one in Fig. 3, it is worth mentioning that even if the former shows the complete system, the diagram is more difficult to read with all its details. As we focus on the development of a certain part of the system, we particularly want to concentrate on visualising that part. This is of high importance especially when the system develops into a significant sized one. Hence, the progress diagram shows the relevant changes in a more legible way.

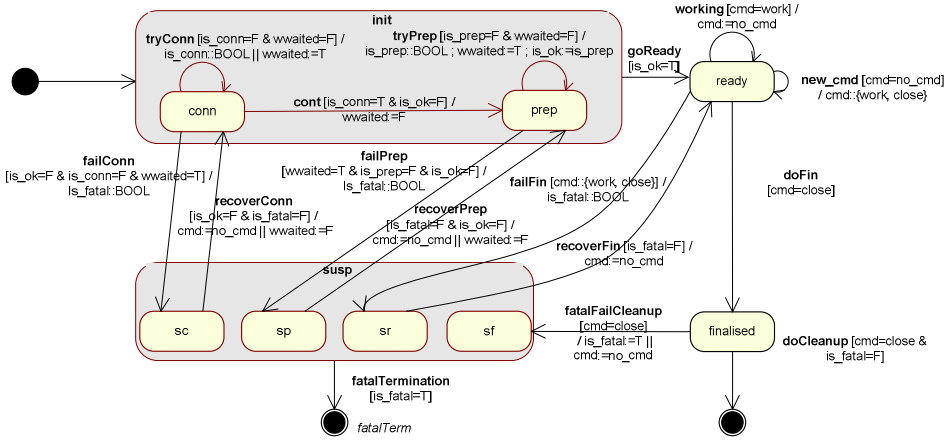

Fig. 4. Statechart diagram of the first refinement step of Memento

In *the second refinement step* (not shown) new hierarchical substates are added in the state *prep* along with new transitions that make use of them. These hierarchical substates indicate that the preparation phase is actually composed of two phases (program as well as module preparation). This step is similar to the one above and is not further described here.

The third refinement step (Fig. 5) strengthens the guards of the transitions (events) (using the choice symbol - salmiakki [15]) and shows a more detailed failure management. In fact we split transitions into alternative paths using the choice points. Each path represents a separate transition whose guard is the conjunction of all the segment guards of that path [15]. Hence, new variables, concerning communication with the server, are introduced to express the details of the program preparation phase. These variables represent sending the identification data (*idDataSent*), reading the response (*respRead*), and checking whether the values for response and user are valid (*respValid* and *userValid*). Furthermore, new failure transitions *nIdDS*, *nRR*, *nRV* and *nUV* corresponding to these variables refine the old general failure transition.

Description	States	Ref. States	Transitions	Ref. Trans.	New Var.
3rd step – adding alternative paths to transitions	-	-	*FailPrepPr*	*nIdDS, nRR, nRV, nUV*	*idDataSent, respRead, respValid, userValid*

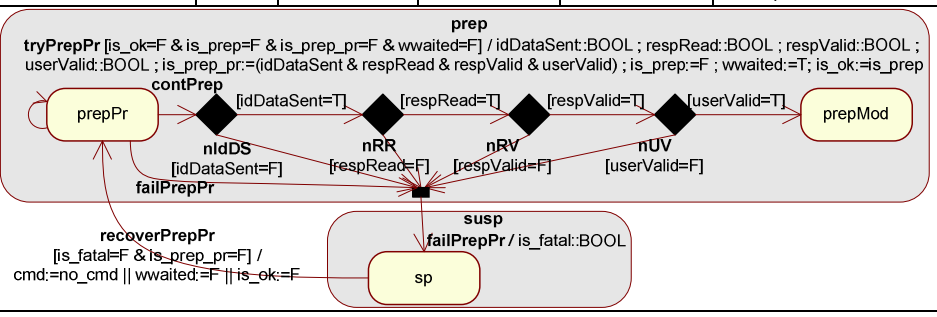

Fig. 5. Third refinement step

Here, the progress diagram also gives an intuitive representation of the proof obligations, now concerning strengthening the guards of the old events (Proof Obligation (2)). This is indicated by the transitions between the *salmiakki* symbols [15] in the diagram part of the progress diagram. Moreover, the outgoing transitions of these symbols illustrate intuitively that the relative deadlock freeness (Proof Obligation (5)) is preserved. Again the partitioning of the error detection event *failPrepPr* in the columns "Transitions" and "Refined Transitions" visualises Proof Obligation (6).

5. Conclusion

This paper presents a new approach to documentation of the stepwise refinement of a system. Since the specification for each step becomes more and more complex and a clear overview of the development is lacking, we focus our approach on illustrating the development steps. This kind of documentation is not only helpful for the developers, but also for those that later will try to reuse the exploited features. The documentation is also useful for communicating the development to stakeholders outside of the development team. Thus, a clear and compact form of progress diagrams is appropriate both for industry developers and researchers.

Formal methods and verification techniques are used in the general design of the Memento application to ensure that the development is correct. Our approach uses the B Method as a formal framework and allows us to address modelling at different levels of abstraction. The progress diagrams give an overview of the refinement steps and the needed proofs. Furthermore, the use of progress diagrams during the incremental construction of large software systems helps to manage their complexity and provides legible and accessible documentation.

In future work we will further explore the link between the progress diagrams and patterns. We will investigate how suitable the progress diagrams are for identifying and differentiating patterns used in the refinement steps. Although progress diagrams already appear to be a viable graphical view of the system development, further experimentation on other case studies is envisaged leading to possible enhancements of the progress diagrams. Tool support will be developed for drawing progress diagrams and linking their analysis with the refined models.

Acknowledgements
We would like to thank Dr Linas Laibinis and Dubravka Ilič for the fruitful discussions on the use of the tools supporting the research.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
[2] J. Arlow and I. Neustadt. *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML.* Addison-Wesley, 2004.
[3] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In: *Proc. of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 131-142, 1983.

[4] R.J.R. Back and K. Sere. From modular systems to action systems. *Software - Concepts and Tools* 17, pp. 26-39, 1996.
[5] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language - a Reference Manual*. Addison-Wesley, 1998.
[6] E. Chan, K. Robinson and B. Welch. Patterns for B: Bridging Formal and Informal Development. In *Proc. of 7th International Conference of B Users (B2007): Formal Specification and Development in B*, LNCS 4355, pp. 125-139, 2007. Springer.
[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.
[8] D. Ilič and E. Troubitsyna. A Formal Model Driven Approach to Requirements Engineering. TUCS Technical Report No 667, Åbo Akademi University, Finland, February 2005.
[9] S.M. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337-356, April 1993.
[10] C. Metayer, J.R. Abrial and L. Voisin. *Event-B Language*, RODIN Deliverable 3.2 (D7), http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf (May 2005)
[11] Object Management Group. *Unified Modelling Language Specification - Complete UML 1.4 specification*, September 2001. http://www.omg.org/docs/formal/01-09-67.pdf
[12] Object Management Group Systems Engineering Domain Special Interest Group (SE DSIG). S. A. Friedenthal and R. Burkhart. *Extending UML™ from Software to Systems*. (accessed 04.05.2007) http://www.syseng.omg.org/
[13] M. Olszewski and M. Pląska. *Memento system*. http://memento.unforgiven.pl, 2006.
[14] C. Snook and M. Butler. U2B - a tool for translating UML-B models into B. In *UML-B Specification for Proven Embedded Systems Design*, chapter 5. Springer, 2004.
[15] C. Snook and M. Waldén. Refinement of Statemachines using Event B semantics. In *Proc. of 7th International Conference of B Users (B2007): Formal Specification and Development in B*, Besançon, France, LNCS 4355, January 2007, pp. 171-185. Springer.
[16] E. Troubitsyna. *Stepwise Development of Dependable Systems*. Turku Centre for Computer Science, TUCS, Ph.D. thesis No. 29. June 2000.
[17] M. Waldén and K. Sere. Reasoning About Action Systems Using the B-Method. *Formal Methods in Systems Design 13(5-35),* 1998. Kluwer Academic Publishers.