

# Documenting the Progress of the System Development\*

Marta Płaska<sup>1</sup>, Marina Waldén<sup>1</sup> and Colin Snook<sup>2</sup>

<sup>1</sup> Åbo Akademi University/TUCS, Joukahaisenkatu 3-5A, 20520 Turku, Finland

<sup>2</sup> 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, Refinement Patterns, 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 [11] 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 [22]. 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 [18, 19] 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, 10]. 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

---

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

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

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 systems and gives refinement patterns and Section 5 introduces the idea of progress diagrams. The system development is analysed and illustrated with the progress diagrams relying on the case study in Section 6. The related work is presented in Section 7. We conclude with some general remarks and our future work in Section 8.

## 2. Case Study – Memento Application

The *Memento* application [14] 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 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 data loss, 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 is used within the development of the system to show the interactions between objects and the order in which these interactions occur. The diagram can be derived directly from the requirements. Furthermore, it may 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 through an internet connection. The second action requires communication with user 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 explained in Section 6).

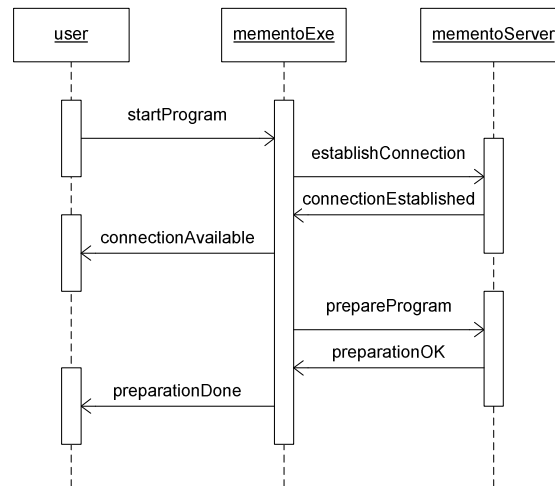


Fig. 1. Sequence diagram presenting the object interaction in the *initialisation phase*

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 a graphical description of the transitions of an object from one state to another in response to events [12, 13]. 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. It 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.

We use the following notation for the transitions in statechart diagrams and in the Event-B code in the rest of the paper. The symbols ‘::’ and ‘:∈’ stand for non-deterministic assignment and are applied interchangeably, in the diagrams and the code, respectively. The symbol ‘:=’ is used in assignments, whereas ‘||’ symbol denotes that the operands are executed concurrently. All of the mentioned symbols are placed in the statement parts. By the use of a junction pseudo state [20] (marked with angled brackets ‘<>’) that denotes the old, refined transition, we indicate the refinement relation between new transitions and previous abstract ones.

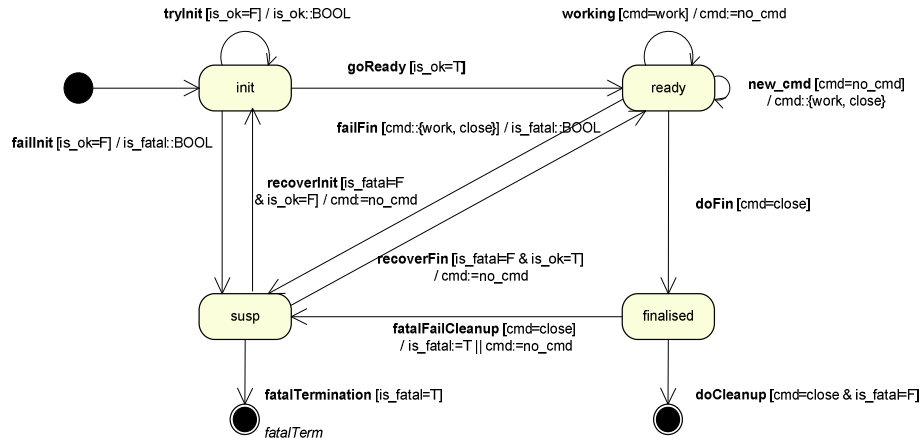


Fig. 2. The abstract statemachine of Memento

### 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 [11]. 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. If several events are enabled simultaneously any one of them may be chosen non-deterministically for execution. 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 [11].

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:

```

MACHINE      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 are then proved using the Event-B prover tool.

### 4. System Refinement and Refinement Patterns

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, 11, 22], 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.

#### 4.1. Refinement of the System

System  $C$  is said to be a correct refinement of  $A$  if the following proof obligations are satisfied [11, 20, 22]:

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 [21]).

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.

#### 4.2. Modelling Refinement Patterns

In order to guide the refinement process and make it more controllable, refinement patterns [12] can be applied. We are using the following notation for the patterns in the rest of the paper. A typical event consists of a guard  $G(V)$  and an action  $S(V)$ , where  $G(V)$  is some supplementary predicate on the variables  $V$ , often represented as a conjunction of several individual guards, and  $S(V)$  is some supplementary assignment of the variables  $V$ . The variables  $V$  are of a general type (TYPE). For instance, in the Event-B code for the refinement EX3c  $G_i(y)$  denotes a guard on variable  $y$  of the general type TYPE, while  $S_i(y)$  denotes some assignment of variable  $y$ .

In all the pattern diagrams (except in the choice paths in Fig. 5) we omit the guards on the transitions for better readability of the diagrams. The code added in the current refinement step is indicated by a darker background.

#### 4.2.1 Refining the States

Let us first concentrate on the abstract specification given in Fig. 3a. It is pictured by a statechart diagram consisting of two states (*st1* and *st2*), a self transition *tr1* for the state *st1* and a transition *tr2* from state *st1* to the state *st2*. We are focusing on data refinement and event refinement patterns enabled by the data refinement. The former is shown in the statechart diagram in Fig. 3b (splitting states into substates and adding transitions between them), while an example of the latter is given in Fig. 3c (splitting existing transitions).

Splitting the states and adding new transitions are commonly performed in one refinement step. The two steps shown in Figures 3b and 3c are shown separately here only to depict the details of the complete data and event refinement. Generally, when refining the states, we want to add some new features/variables at the same time as we split the transitions.

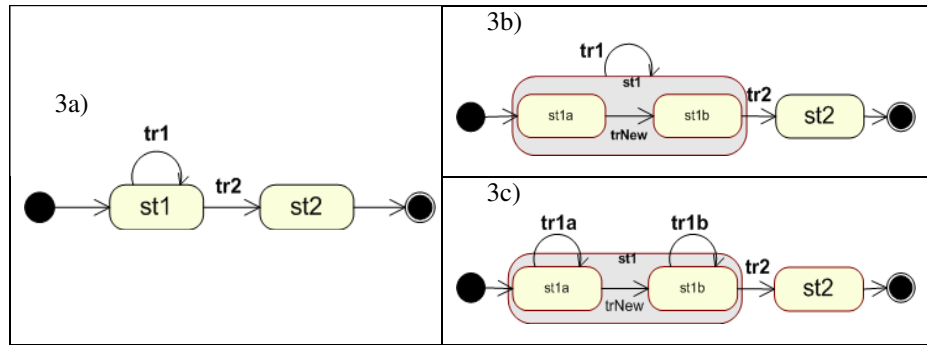


Fig. 3. Refinement patterns – basic data and event refinements

For the abstract specification depicted in Fig. 3a we have the following Event-B specification code:

```

MACHINE      EX3a
VARIABLES    state
INVARIANT    state ∈ {st1, st2}
INITIALISATION state:=st1
EVENTS
  tr1 =        WHEN state=st1 THEN state:=st1 END;
  tr2 =        WHEN state=st1 THEN state:=st2 END
END

```

The Event-B code for the pattern concerning the data refinement, i.e. splitting the states, is illustrated in the Fig. 3b as follows:

```

REFINEMENT    EX3b
REFINES       EX3a
VARIABLES    state, state1
INVARIANT    state ∈ {st1, st2} ∧ state1 ∈ {st1a, st1b}
INITIALISATION state:=st1 || state1:=st1a
EVENTS
  tr1 =        WHEN state=st1 THEN state:= st1 END;
  trNew =      WHEN state=st1 ∧ state1=st1a THEN state1:=st1b END;
  tr2 =        WHEN state=st1 ∧ state1=st1b THEN state:=st2 END
END

```

The pattern for separating an existing transition (event refinement) corresponding to the diagram in Fig. 3c is as follows:

```

REFINEMENT    EX3c
REFINES      EX3a
VARIABLES    state, state1, y, z
INVARIANT    state ∈ {st1, st2} ∧ state1 ∈ {st1a, st1b} ∧ y ∈ TYPE ∧ z ∈ TYPE ∧ I(y,z)
INITIALISATION state:=st1 || state1:=st1a || y:∈TYPE || z:∈TYPE
EVENTS
  tr1a (refines tr1) = WHEN state=st1 ∧ state1=st1a ∧ G1a(y)
                       THEN state:=st1 || state1:=st1a || S1a(y) END;
  tr1b (refines tr1) = WHEN state=st1 ∧ state1=st1b ∧ G1b(z)
                       THEN state:=st1 || state1:=st1b || S1b(z) END;
  trNew =           WHEN state=st1 ∧ state1=st1a ∧ GN(y)
                       THEN state1:=st1b END;
  tr2 =             WHEN state=st1 ∧ state1=st1b ∧ G2(z)
                       THEN state:=st2 END
END

```

Each of the refined event uses some of the new variables ( $y$  and  $z$ ) in its guards ( $G(y)$  and  $G(z)$ ) and actions ( $S(y)$  and  $S(z)$ ) to reduce non-determinism. The guards  $G_{1a}(y)$  and  $G_N(y)$  are created in such a way that they guarantee progress. In the same manner  $G_N(y)$  should imply  $G_2(z)$  or  $G_{1b}(z)$ , moreover guards  $G_2(z)$  and  $G_{1b}(z)$  should also be formed to guarantee the progress of the system. When inserting conditions on new properties to the guards, the failure management is in general also refined in a corresponding manner in order to design a fault tolerant behaviour. Nevertheless, here we concentrate our patterns on the proper and desirable behaviour of the system. An example of another pattern of the basic data and event refinement including failure management is given by Snook and Waldén [20]. In that pattern a loop is created in the superstate.

In order to give an intuition of the correctness of the patterns, we state how the Proof Obligation Rules given in Section 4.1 are satisfied by the patterns. Moreover, the Proof Obligations hint at how problems in the program design can be detected more easily.

According to the Proof Obligations (1) and (2) in Section 4.1 the initialisation and the events are refined to take the new variables into consideration. The guards of the old events may be strengthened and assignments concerning the new variables added. During the system development we may also want to refine an existing event by splitting it into several separate events. As acknowledged in Proof Obligation (3), the new events are only permitted to assign the new variables, but may, however, refer to the old variables. The guard of the new event should be composed in such a way that the new event, together with the refined events, ensures progress of the system (Proof Obligation (5)).

The new events should not take over the execution (Proof Obligation (4)), which can be assured by disallowing the new transitions in the statemachine diagram (corresponding to new events in the Event-B model) from forming a loop. The Event-B prover requires an expression, called a ‘variant’, that gives a Natural number that is decreased by all of the new transitions between the substates. To deduce a suitable variant, graph theory is applied. The states of the statemachine representing the system are numbered according to the minimum path length to a refining transition. Hence, if the new transitions form a sequence that progresses towards a refining transition, the function that defines this numbering for each state will be strictly

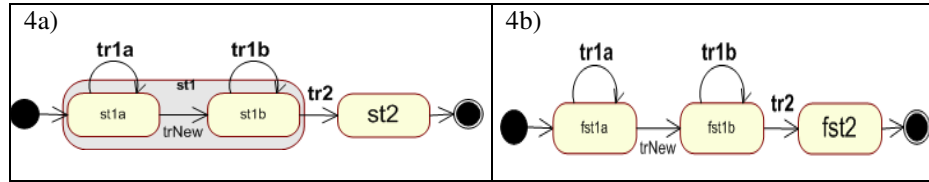


decreased as the state changes. If loops are unavoidable in the new events, the auxiliary variables must be used in the variant in order to provide a suitable variant that decreases throughout the loop. Each new transition has to lead to a new state with a lower designated number or (in case of loops) alter the auxiliary state variables, thereby decreasing the variant. To avoid deadlock, a route from every new transition to one that refines an old transition should exist. This is a necessary, but not sufficient, condition for relative deadlock freeness. If there exists no sequence of new transitions which can reach one that refines an old transition, meaning there is no route, then new events terminate (without enabling an old event) and a new deadlock is introduced [20].

As properties are added to the system, the potential failure management should also be refined. If a fault appears at a substate it should be viable to return to that substate after recovery. This can be achieved by dividing an abstract failure into more specific failures on the new features in conformance with Proof Obligation (6). Note that new failure situations are not introduced in our case, as it is a general pattern that can be adjusted to the specific needs.

#### 4.2.2 Flattening States

When refining the system by superposition and at the same time splitting the states in a hierarchical manner, we have to deal with the states that are nested in the superstate due to the consequent refinement steps. This development, although performed in a stepwise manner, at some point makes the system model unreadable. Therefore we apply the flattening pattern, which removes the most external superstate, leaving the substates intact.



**Fig. 4.** Refinement pattern – flattening of the hierarchical states

In Fig. 4 we present the flattening pattern applied to the model from Fig. 3c. In Fig. 4a we model the hierarchical structure of states, i.e. state *st1* is a superstate for the states *st1a* and *st1b*. By applying the flattening pattern, we remove the superstate *st1*. This is possible, when giving an appropriate invariant preserving the relation between the states in the model, namely relating the states from the old model to the states in the new model. This is correlated with the change of the naming of the variables in order to preserve the invariant.

Note that flattening can only be performed once the parent state is neither the source nor target of any transitions. That is, other patterns should first be applied to move all the parents' transitions to its substates so that the parent state is completely redundant.

Here we show the Event-B code for the refined model:

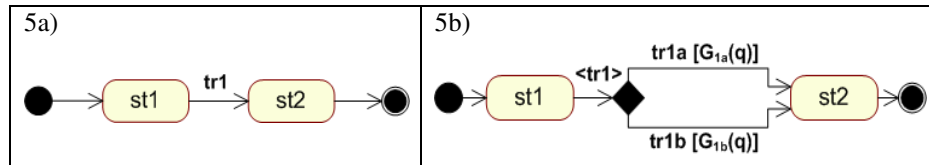
<b>REFINEMENT</b>	EX4b
<b>REFINES</b>	EX4a
<b>VARIABLES</b>	newState, y, z, v
<b>INVARIANT</b>	$\text{newState} \in \{\text{fst1a}, \text{fst1b}, \text{fst2}\} \wedge \text{newState} \in \text{NEWSTATE}$ $\wedge y \in \text{TYPE} \wedge z \in \text{TYPE} \wedge v \in \text{TYPE} \wedge R_4(y, z, v) \wedge$ $\text{newState} = \text{fst1a} \Leftrightarrow (\text{state} = \text{st1} \wedge \text{state1} = \text{st1a}) \wedge$ $\text{newState} = \text{st1b} \Leftrightarrow (\text{state} = \text{st1} \wedge \text{state1} = \text{st1b}) \wedge \text{state} = \text{st2} \Rightarrow \text{newState} = \text{fst2} \wedge$ $\text{newState} = \text{fst2} \Rightarrow \text{state} = \text{st2} \wedge \text{state} = \text{st1} \Rightarrow (\text{newState} = \text{fst1a} \vee \text{fst1b})$ $\text{newState} = \text{fst1a} \parallel v \in \text{TYPE} \parallel y \in \text{TYPE} \parallel z \in \text{TYPE}$
<b>INITIALISATION</b>	
<b>EVENTS</b>	
tr1a (refines tr1) =	<b>WHEN</b> $\text{newState} = \text{fst1a} \wedge G_{1a}(y) \wedge G_{1a}(z) \wedge G_{1a}(v)$ <b>THEN</b> $\text{newState} := \text{fst1a} \parallel S_{1a}(y) \parallel S_{1a}(z) \parallel S_{1a}(v)$ <b>END</b> ;
tr1b (refines tr1) =	<b>WHEN</b> $\text{newState} = \text{fst1b} \wedge G_{1b}(y) \wedge G_{1b}(z) \wedge G_{1b}(v)$ <b>THEN</b> $\text{newState} := \text{fst1b} \parallel S_{1b}(y) \parallel S_{1b}(z) \parallel S_{1b}(v)$ <b>END</b> ;
trNew =	<b>WHEN</b> $\text{newState} = \text{fst1a} \wedge G_{nr}(y) \wedge G_{nr}(z) \wedge G_{nr}(v)$ <b>THEN</b> $\text{newState} := \text{fst1b} \parallel S_{nr}(y) \parallel S_{nr}(z) \parallel S_{nr}(v)$ <b>END</b> ;
tr2 =	<b>WHEN</b> $\text{newState} = \text{fst1b} \wedge G_{2r}(y) \wedge G_{2r}(z) \wedge G_{2r}(v)$ <b>THEN</b> $\text{newState} := \text{fst2} \parallel S_{2r}(y) \parallel S_{2r}(z) \parallel S_{2r}(v)$ <b>END</b>
<b>END</b>	

Since flattening the state hierarchy is rather a rewriting step than a refinement step, the proof obligations in Section 4.1 trivially hold. We rely on the invariant giving the relation between the flattened state and the hierarchical states.

#### 4.2.3 Separating Existing Transitions – Choice Paths

In order to perform event refinement, particularly to separate existing events, we can split the transition into alternative paths using the black diamond-shaped choice symbol (salmiakki) [20], where each choice is responsible for a separate event. The guards on the events are strengthened by the choice points. Each choice represents a separate event whose guard includes the conjunction of all the segments leading up to that path. Thus, the guard enabling a given event is the conjunction of all the conditions of choice paths leading up to the choice point.

Fig. 5 illustrates a simple pattern for adding features to the specification and expanding its functionality. More specifically, the transition *tr1* is refined by two branches - transitions *tr1a* and *tr1b*. This could, for example, model the refinement of a non-deterministic event ‘*move*’ to the more specific events ‘*move\_forward*’ and ‘*move\_backward*’.



**Fig. 5.** Refinement pattern – event refinement: simple choice paths

The Event-B code corresponding to this pattern for the abstract machine is as follows:

<b>MACHINE</b>	EX5a
<b>VARIABLES</b>	state
<b>INVARIANT</b>	$\text{state} \in \{\text{st1}, \text{st2}\}$
<b>INITIALISATION</b>	$\text{state} := \text{st1}$
<b>EVENTS</b>	
tr1 =	<b>WHEN</b> $\text{state} = \text{st1}$ <b>THEN</b> $\text{state} := \text{st2}$ <b>END</b>
<b>END</b>	

The refinement can be expressed as the following Event-B machine:

```

REFINEMENT    EX5b
REFINES      EX5a
VARIABLES    state, q
INVARIANT    state ∈ {st1, st2} ∧ q ∈ TYPE ∧ I(q)
INITIALISATION state:=st1 || q := TYPE
EVENTS
  tr1a (refines tr1) = WHEN state=st1 ∧ G1a(q) THEN state:=st2 || S1a(q) END;
  tr1b (refines tr1) = WHEN state=st1 ∧ G1b(q) THEN state:=st2 || S1b(q) END
END

```

Guards of the transition *tr1* are strengthened via choice point, according to Proof Obligation (2). When splitting the transition *tr1* into two more specific transitions *tr1a* and *tr1b* we should ensure the progress of the system, fulfilling Proof Obligation (5).

With the choice paths pattern we can also show more detailed failure management. Fig. 6 depicts the creation of the choice path on the transition *tr1*, thus detailing the error detection. The transition *tr1* is refined by strengthening its guards concerning the new variable *q* ( $G_I(q)$ ). The transition *tr2undef* is refined into transition *tr2undef2*, which stands for the detection of undetermined errors and *tr2a* which is modelling a particular type of failures ( $\neg G_I(q)$ ).

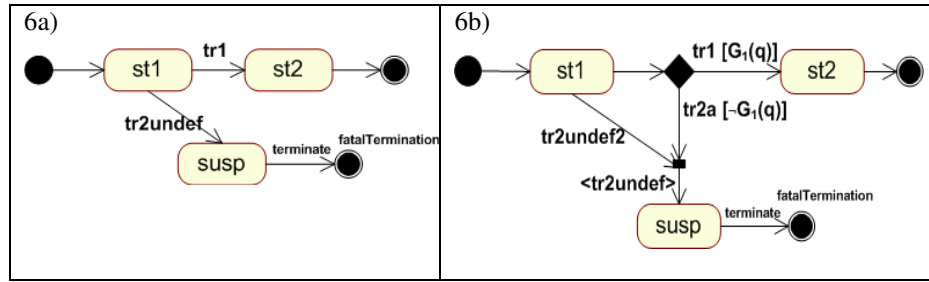


Fig. 6. Refinement pattern – event refinement: choice paths

The Event-B code for the abstract machine that we use as an example for choice paths pattern (event refinement) is as follows:

```

MACHINE      EX6a
VARIABLES    state
INVARIANT    state ∈ {st1, st2, susp}
INITIALISATION state:=st1
EVENTS
  tr1 = WHEN state=st1 THEN state:=st2 END;
  tr2undef = WHEN state=st1 THEN state:=susp END
END

```

The refined model is expressed as an Event-B model given below:

```

REFINEMENT    EX6b
REFINES      EX6a
VARIABLES    state, q
INVARIANT    state ∈ {st1, st2, susp} ∧ q ∈ TYPE ∧ J(q)
INITIALISATION state:=st1 || state1:=st1a || q := TYPE
EVENTS
  tr1 = WHEN state=st1 ∧ G1(q) THEN state:=st2 || S1(q) END;
  tr2undef2 (refines tr2undef) = WHEN state=st1 THEN state:=susp END;
  tr2a (refines tr2undef) = WHEN state=st1 ∧ ¬G1(q) THEN state:=susp || S2a(q) END
END

```

We use the join (black bar symbol) to illustrate refinement of the failure transition *tr2undef*, which is split into two different failures, *tr2undef2* and *tr2a*, in accordance with Proof Obligation (6). The guard for transition *tr1* is strengthened (Proof Obligation (2)) by the conjunction of the negation of all the particular failures. In this way we can ensure that there will be an enabled event also in the refined model (Proof Obligation (5)).

#### 4.2.4 Orthogonal Regions Pattern

Furthermore, we can also consider a pattern for adding the same behaviour to several states (orthogonal regions [20]) as a type of data and event refinement. This pattern can be used in case of architectural redundancy, i.e. when several states have incoming (entry) and outgoing (exit) transitions of similar functionality. Fig. 7 illustrates adding an orthogonal region (the lower region) to the superstate *susp*, which has new behaviour common to all the previous states (given in the higher region), applicable to all three kinds of failure. In order for the pattern to be correct, several conditions have to be fulfilled. The orthogonal region should not affect the mechanism of error detection. In Fig. 7 we show that the unnamed entry and exit transitions of the lower region connect to the named events of the upper region. It must be ensured that, when the new region is entered, at least one of the new transitions must be enabled (synchronisation condition). Moreover the exit transitions from the upper region are synchronised with equivalent transitions of the lower region, i.e., they are guarded by the lower region reaching a state that has an exit transition with which it can merge.

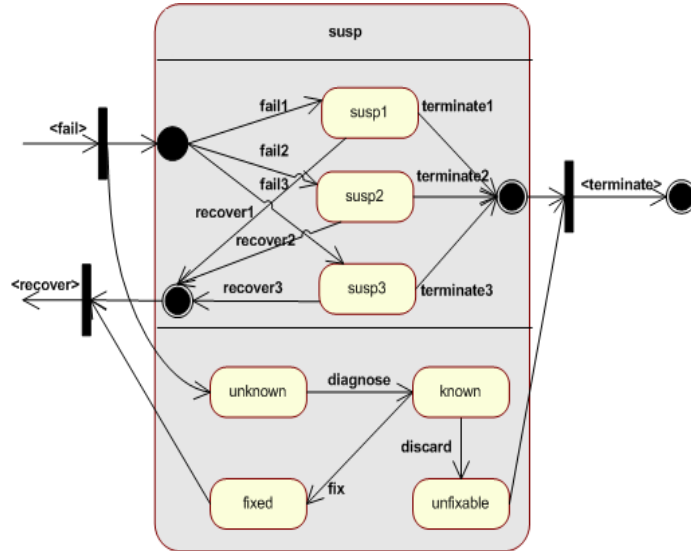


Fig. 7. Refinement pattern - superposition of an orthogonal region

Subsequently we give a machine and its refinement using the orthogonal states. The upper region of the state *susp* forms the abstract machine.

```

MACHINE      EX7
VARIABLES    state, suspState
INVARIANT    state ∈ {susp, state_ok} ∧ suspState ∈ {susp1, susp2, susp3}
INITIALISATION state:=susp || suspState:={susp1, susp2, susp3}
EVENTS
  fail1 =      WHEN state=state_ok                THEN state:=susp || stateSusp:=susp1 END;
  recover1 =    WHEN state=susp ∧ suspState=susp1    THEN state:=state_ok                END;
  terminate1 =  WHEN state=susp ∧ suspState=susp1    THEN state:=fatalTermination        END;
  ...
END

```

In the refinement the old state *susp* is composed with the orthogonal region.

```

REFINEMENT    EX7a
REFINES       EX7
VARIABLES    state, suspState, ortState, s
INVARIANT    state ∈ {susp, state_ok, fatalTermination} ∧ suspState ∈ {susp1, susp2, susp3} ∧
               ortState ∈ {unknown, known, fixed, unfixable} ∧ s ∈ TYPE ∧ l(s)
INITIALISATION state:=susp || suspState:={susp1} || ortState:=unknown || s:∈ TYPE
EVENTS
  fail1 =      WHEN state=state_ok ∧ Gr(s)
               THEN state:=susp || stateSusp:=susp1 || ortState:=unknown END;
  recover1 =    WHEN state=susp ∧ suspState=susp1 ∧ Gr(s) ∧ ortState=fixed
               THEN state:=state_ok || Sr(s) END;
  terminate1 =  WHEN state=susp ∧ suspState=susp1 ∧ ¬Gr(s) ∧ ortState=unfixable
               THEN state:=fatalTermination || Sr(s) END;
  ...
  diagnose =    WHEN state=susp ∧ ortState=unknown    THEN ortState:=known END;
  discard =     WHEN state=susp ∧ ortState=known      THEN ortState:=unfixable END;
  fix =         WHEN state=susp ∧ ortState=known      THEN ortState:=fixed || Sr(s) END
END

```

As the events/transitions introduced in the orthogonal region are new events in the refinement, they should only concern new features to satisfy Proof Obligation (3). These new transitions should eventually hand over control to the old transitions, which is guaranteed by Proof Obligation (4). Hence, we need to generate a variant on the distance to an *exit* transition for these new transitions. In order to fulfil Proof Obligation (5) at least one of the new transitions (*diagnose* followed by *discard* or *fix*) must be enabled after entering the orthogonal region. This is caused by the fact that the recovering transitions and the terminating transitions wait to be enabled until the orthogonal region is prepared to synchronise with them. The composed events are then refinements of the old events, like for example *fail1*, *fail2*, *fail3*, in conformance with Proof Obligation (2). The orthogonal region strengthens the guards of the termination and recovery events, but at the same time it guarantees that an exit state of the orthogonal region will be reached (Proof Obligation (6)).

As the size of the system grows during the development, it is difficult to get a clear overview of the refinement process. In this paper we benefit from *progress diagrams* [16] to give an abstraction and graphical-descriptive view documenting the applied patterns in each step. The pattern types are illustrated in more detail with the progress diagrams to show the relevant development changes in a legible manner. This is of high importance especially when the system evolves into a significantly sized one.

## 5. Progress Diagrams

We exploit the progress diagram [16], which is in the form of a table 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 are refined, as well as new variables that are added with respect to these features. Progress diagrams do not involve any mathematical notation and are, therefore, useful for communicating the development steps to non-formal methods colleagues.

Event-B classifies events as ‘convergent’ if they are new events that are expected to eventually relinquish control to an old (refined) event (i.e. it must decrease the variant). Events that are not convergent are classified as ordinary. A third classification, ‘anticipating’, refers to events that will be shown to be convergent in a future refinement, but we do not use such events in the examples of the patterns.

We call the transition that starts a sequence of convergent transitions an ‘initiator’. To ensure feasibility of the sequence of transitions, the guard of an initiator must imply the guard of at least one of the old transitions that it could lead to. We call the transition ‘refined’ if it refines an existing transition according to the refinement rules (given in Section 4.1 of the paper), leaving the system in an equivalent state to the post-state of the transition being refined.

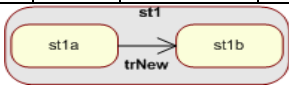
We envisage a tool which will automatically create a new refinement from the progress diagram. In order to be able to design and implement such a tool, we extend the tabular part of the progress diagram to provide the required information. As a result, we add new features in the Refined Transitions part indicating the source and target state of the refined transition, as well as the initialisation of the variables added in the current refinement step. The diagram part gives a supplementary view of the current refinement step and is, in fact, a fragment of the statechart diagram. It can be used as assistance for the developer and to support the documentation.

During the development we profit 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 that we are most interested in, and provides a legible picture of the (possibly complex) system development. The visualisation helps us to better understand the refinement steps and proofs that need to be performed.

When proving the refined system, the progress diagram indicates the needed proof obligations. If new states (column “Ref. States”) and variables (column “New Var.”) are added, they should be initialised according to the invariant (Proof Obligation (1)). In the progress diagram the refined events are given in the column “Refined Transitions” and have a corresponding event in the column “Transitions” (Proof Obligation (2)). Also the convergent events are given in the column “Refined Transitions”. 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

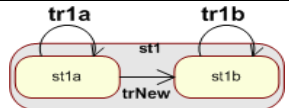
the convergent transitions (Proof Obligation (4)) is indicated in the diagram part by the fact that these transitions do not form a loop. The columns “Transitions” and “Refined Transitions” also illustrate partitioning of the error detection events (Proof Obligation (6)). The progress of the refined specification always has to be ensured in line with Proof Obligation (5).

In order to illustrate the idea of progress diagrams in combination with refinement patterns, we use the abstract system (shown in Fig. 3a, Section 4.2.1) consisting of two states ( $st1$  and  $st2$ ) and two transitions ( $tr1$  and  $tr2$ ). We refine it to the concrete system shown in Fig. 3b, where the state ( $st1$ ) is partitioned into substates ( $st1a$  and  $st1b$ ) and the anticipating transition  $trNew$  is added between the new substates. The progress diagram of this sample refinement step is depicted in Fig. 8.

Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
1 <sup>st</sup> refinement step: • creating hierarchical substates (in state $st1$ ) – data ref. • adding new transition concerning the substates ( $trNew$ ) – event ref.	$st1$	$st1a$ $st1b$	-	$trNew$ ( $st1a, st1b$ )	-
					

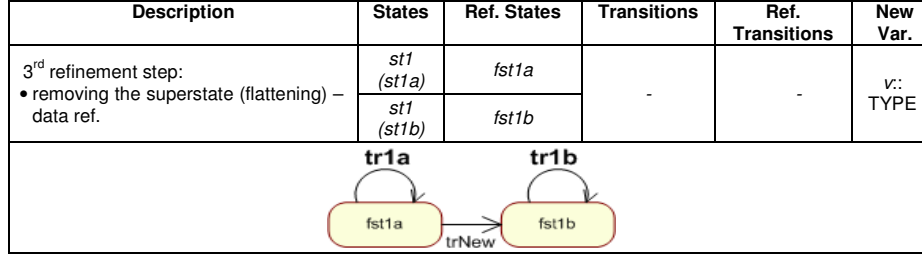
**Fig. 8.** Example of a progress diagram for the pattern 3b from Section 4

In the progress diagram in Fig. 9 we depict the event refinement by separating existing transitions. We continue refining the system shown in Fig. 3b in Section 4.2.1, by detailing its functionality and splitting the existing self-transition  $tr1$  into self-transitions  $tr1a$  and  $tr1b$ , according to the substates separated in the previous step. We also assume that with respect to the added transitions in the refined system we simultaneously add new variables  $y$  and  $z$ . The new variables and their initialisation are depicted in the rightmost column of the progress diagram.

Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
2 <sup>nd</sup> refinement step: • separating the existing transition $tr1$ into two transitions ( $tr1a$ and $tr1b$ ) concerning the substates ( $st1a$ and $st1b$ ) – event ref.	-	-	$tr1$	$tr1a$ ( $st1a, st1a$ ), $tr1b$ ( $st1b, st1b$ )	$y::$ TYPE $z::$ TYPE
					

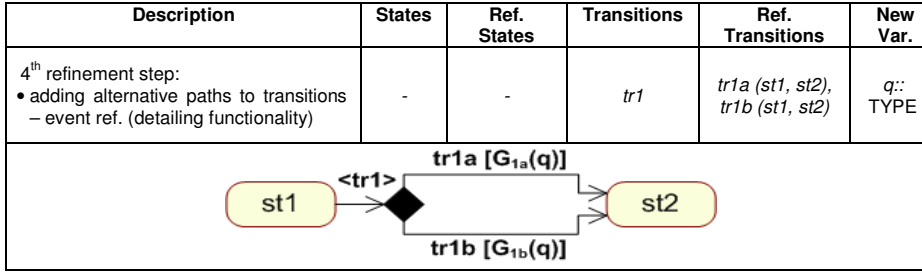
**Fig. 9.** Example of a progress diagram for the pattern 3c from Section 4

We also consider the flattening pattern to diminish complex hierarchical state structure created while performing consecutive refinement steps. In Fig. 10 we show the progress diagram for the flattening pattern for the diagram in Fig. 3c given in Section 4.2.2.



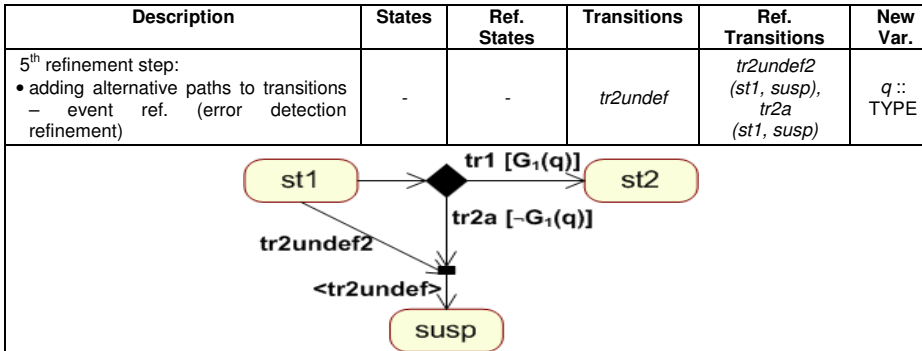
**Fig. 10.** Example of the progress diagram for the flattening pattern

The progress diagram of the choice path pattern is depicted in Fig.11 and Fig. 12. In Fig. 11 we create a choice path on the transition  $tr1$ , by refining the transition  $tr1$  into two more specific transitions  $tr1a$  and  $tr1b$ . The guards  $G_{1a}(q)$  and  $G_{1b}(q)$  are created in such a way that we ensure progress of the system.



**Fig. 11.** Example of a progress diagram for the choice paths pattern (specifying functionality)

The splitting could also be used to model more detailed failure. In Fig. 12 we split the transition  $tr2undef$  between the states  $st1$  and  $susp$  into two transitions,  $tr2a$  and  $tr2undef2$ , by strengthening the guard condition on the refined transition  $tr2a$ . The guard of the refined transition  $tr1$  is the negation of the refined failure transition  $tr2a$ .



**Fig. 12.** Example of a progress diagram for the choice paths pattern (specifying error detection)

When adding common behaviour to the existing states (superposition of an orthogonal region), we want to express which of the new transitions are performing the functionality of the old ones. Therefore, we refine existing system (shown in the



upper part of the superstate in Fig. 7) to a functionally more structured and unified one (shown in the lower part of the superstate in Fig. 7). Thereby we create a behavioural pattern, where the system before the refinement is synchronised with the system after the refinement.

In the progress diagram in Fig. 13 we depict the orthogonal region as follows. We show only the lower region without the previous higher region in the superstate. In the tabular part we compare the old superstate with the new one using a bracket notation (superstate {subA1, subA2...} {subB1, subB2...}) to indicate the hierarchy of states in regions. We indicate the states that need synchronisations with existing incoming and outgoing transitions. Furthermore, we specify the new transitions in the orthogonal region and add some variables according to the refinement step.

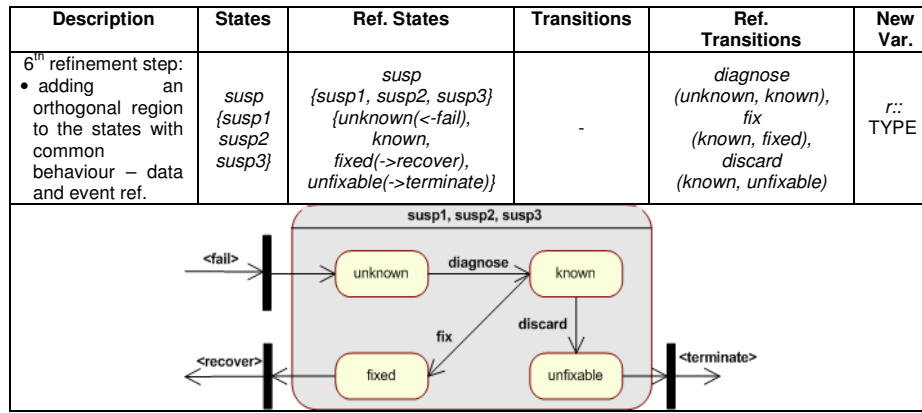
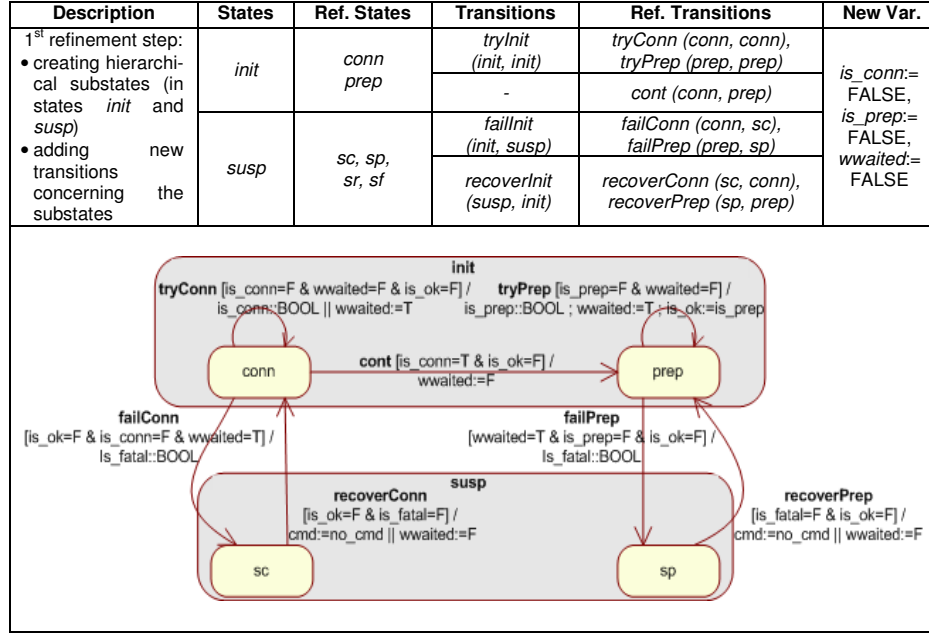


Fig.13. Example of progress diagram for the orthogonal region pattern

## 6. Case Study Memento

Fig. 14 depicts the progress diagram of the *first refinement step* for the Memento system (following the abstract specification presented in Section 3), 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 phase (state *conn*) and a preparation phase (state *prep*), that both need 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

initiator, transition *cont* (added between the new substates *conn* and *prep*), converges to *tryPrep* and *failPrep* which are refined transitions of *tryInit* and *failInit* respectively. The initiator is guarded by the guard (*is\_ok=FALSE*) from *tryInit*, thus ensuring feasibility. New variables *is\_conn*, *is\_prep* and *wwaited* are introduced to control the system execution flow. Note that there are separate diagram parts (not shown) for the substates *sr* and *sf*.



**Fig. 14.** 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 provides an overview of the proof obligations needed for the refinement step. Since we add new states and variables, we indicate that the old transitions and initialisation need to be refined, according to Proof Obligation (1) and (2). For example in Fig. 14 events *tryConn* and *tryPrep* refine *tryInit*. Event *cont* is a convergent event that only assigns the new variable *wwaited* (Proof Obligation (3)). Since this event is the only newly introduced event in this refinement step and it connects two separate states *conn* and *prep*, Proof Obligation (4) is fulfilled. Furthermore, the error detection event *failinit* is partitioned into *failConn* and *failPrep* in line with Proof Obligation (6). The transitions are composed in such a way that they ensure progress in the diagram (Proof Obligation (5)).

The result of the first refinement step is shown in the statechart diagram in Fig. 15. When comparing this diagram to the one in Fig. 14, it is worth mentioning that even if the former shows the complete system, the diagram is more difficult to read with all its details. The progress diagram shows only the relevant changes in a more legible way.

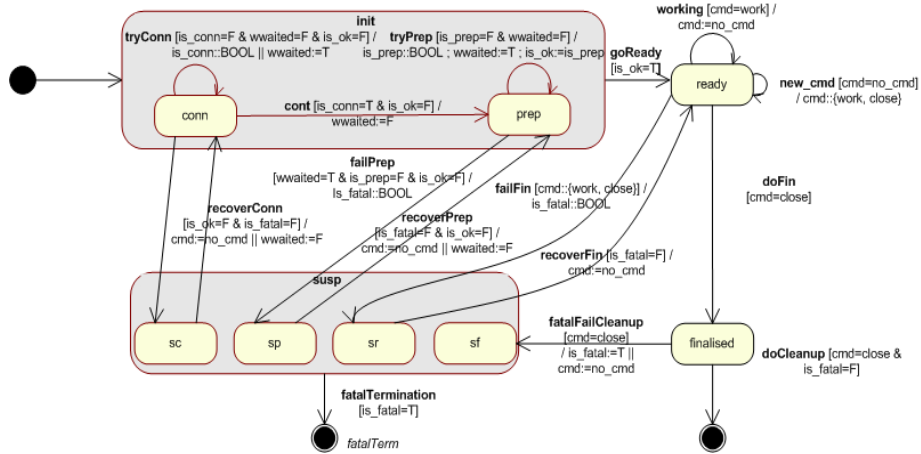


Fig. 15. Statechart diagram of the first refinement step of Memento

The excerpt of Event-B code depicting the first refinement step of Memento system is given below.

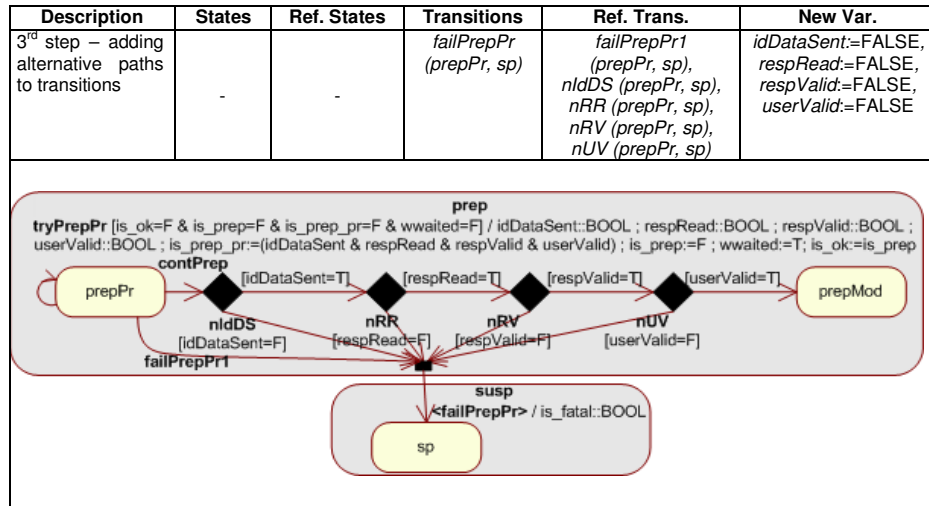
```

REFINEMENT      Memento_Ref
REFINES         Memento
SEES            Data
VARIABLES       is_fatal, is_ok, cmd, state, wwait, is_conn, is_prep, conn, prep, sc, sp
INVARIANT       is_fatal ∈ BOOL ∧ is_ok ∈ BOOL ∧ cmd ∈ CMD ∧ state ∈ STATE ∧
                  (state=init ⇒ cmd=no_cmd) ∧
                  init_state ∈ {conn, prep} ∧ susp_state ∈ {sc, sp} ∧
                  (state=init ∧ init_state=prep ⇒ is_conn=TRUE) ∧
                  (state=susp ∧ susp_state ∈ {sc, sp} ⇒ cmd=no_cmd) ∧
                  (state=susp ∧ susp_state=sp ⇒ is_conn=TRUE) ∧
                  (is_prep=TRUE ⇒ is_conn=TRUE) ∧ ...
INITIALISATION is_fatal:=FALSE || cmd:=no_cmd || is_ok:=FALSE || state:=init ||
                  is_conn:=FALSE || is_prep:=FALSE || wwait:=FALSE || susp_state:=sc ||
                  init_state:=conn
EVENTS
tryConn (refines tryInit) =
  WHEN state=init ∧ init_state=conn ∧ is_ok=FALSE ∧ is_conn=FALSE ∧ wwait=FALSE
  THEN is_conn := TRUE || wwait:=TRUE || is_ok := TRUE END;
failConn (refines failInit) =
  WHEN state=init ∧ init_state=conn ∧ is_ok=FALSE ∧ wwait=TRUE
  THEN state:=susp || susp_state:=sc || is_fatal := TRUE END;
recoverConn (refines recoverInit) =
  WHEN state=susp ∧ susp_state=sc ∧ is_ok=FALSE ∧ is_fatal=FALSE
  THEN state:=init || init_state:=conn || cmd:=no_cmd || wwait:=FALSE END;
tryPrep (refines tryInit) =
  WHEN state=init ∧ init_state=prep ∧ is_ok=FALSE ∧ is_prep=FALSE ∧ wwait=FALSE
  THEN is_prep := TRUE || wwait:=TRUE || is_ok := TRUE END;
failPrep (refines failInit) =
  WHEN state=init ∧ init_state=prep ∧ wwait=FALSE ∧ is_prep=FALSE ∧ is_ok=FALSE
  THEN state:=susp || susp_state:=sp || is_fatal := TRUE END;
recoverPrep (refines recoverInit) =
  WHEN state=susp ∧ susp_state=sp ∧ is_ok=FALSE ∧ is_fatal=FALSE
  THEN state:=init || init_state:=prep || cmd:=no_cmd || wwait:=FALSE END;
goReady =
  WHEN state=init ∧ is_ok=TRUE ∧ is_conn=TRUE ∧ is_prep=TRUE ∧ init_state=prep
  THEN state:=ready END;
...
END

```

In the *second refinement step* 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. 16) strengthens the guards of the transitions/events (according to the pattern in Fig. 6) and shows a more detailed failure management. 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.



**Fig. 16.** 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 *choice point* symbols 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).

Below we show the partial Event-B code for the third refinement step of our case study.

<b>REFINEMENT</b>	Memento_Ref2
<b>REFINES</b>	Memento_Ref1
<b>SEES</b>	Data
<b>VARIABLES</b>	is_fatal, is_ok, cmd, state, wwait, is_conn, is_prep, conn, prep, sc, sp, sr, prepPr, prepMod, prep_pr, prep_mod, is_prep_pr, idDataSent, respRead, respValid, userValid
<b>INVARIANT</b>	is_fatal ∈ BOOL ∧ is_ok ∈ BOOL ∧ cmd ∈ CMD ∧ state ∈ STATE ∧

```

(state=init  $\Rightarrow$  cmd=no_cmd)  $\wedge$ 
init_state  $\in$  {conn, prep}  $\wedge$  susp_state  $\in$  {sc, sp}  $\wedge$ 
(state=init  $\wedge$  init_state=prep  $\Rightarrow$  is_conn=TRUE)  $\wedge$ 
(state=susp  $\wedge$  susp_state  $\in$  {sc, sp, sr}  $\Rightarrow$  cmd=no_cmd)  $\wedge$ 
(state=susp  $\wedge$  susp_state=sp  $\Rightarrow$  is_conn=TRUE)  $\wedge$ 
(is_prep=TRUE  $\Rightarrow$  is_conn=TRUE)  $\wedge$ 
idDataSent  $\in$  BOOL  $\wedge$  respRead  $\in$  BOOL  $\wedge$ 
respValid  $\in$  BOOL  $\wedge$  userValid  $\in$  BOOL  $\wedge$  prep_state:=prep_pr  $\wedge$ 
(state=init  $\wedge$  init_state=prep  $\wedge$  prep_state=prep_mod  $\Rightarrow$ 
  idDataSent=TRUE  $\wedge$  respRead=TRUE  $\wedge$  respValid=TRUE  $\wedge$  userValid=TRUE)  $\wedge$ 
(state=susp  $\wedge$  susp_state=sr  $\Rightarrow$ 
  idDataSent=TRUE  $\wedge$  respRead=TRUE  $\wedge$  respValid=TRUE  $\wedge$  userValid=TRUE)  $\wedge$ 
(is_prep_pr=TRUE  $\Rightarrow$ 
  idDataSent=TRUE  $\wedge$  respRead=TRUE  $\wedge$  respValid=TRUE  $\wedge$  userValid=TRUE) ...
INITIALISATION is_fatal:=FALSE || cmd:=no_cmd || is_ok:=FALSE || state:=init ||
is_conn:=FALSE || is_prep:=FALSE || wwaited:=FALSE || susp_state:=sc ||
init_state:=conn || prep_state:=prep_pr || is_prep_pr:=FALSE || idDataSent:=FALSE
|| respRead:=FALSE || respValid:=FALSE || userValid:=FALSE
EVENTS
tryPrepPr (refines tryPrep) =
  WHEN state=init  $\wedge$  init_state=prep  $\wedge$  is_ok=FALSE  $\wedge$ 
  is_prep=FALSE  $\wedge$  wwaited=FALSE  $\wedge$  is_prep_pr=FALSE  $\wedge$  prep_state=prep_pr
  THEN idDataSent  $\in$  BOOL; respRead  $\in$  BOOL; respValid  $\in$  BOOL; userValid  $\in$  BOOL;
  IF (idDataSent=TRUE  $\wedge$  respRead=TRUE  $\wedge$  respValid=TRUE  $\wedge$  userValid=TRUE)
  THEN is_prep_pr:=TRUE
  ELSE is_prep_pr:=FALSE END;
  is_prep:=FALSE; wwaited:=TRUE; is_ok:=is_prep END;
failPrepPr (refines failPrep) =
  WHEN state=init  $\wedge$  init_state=prep  $\wedge$  wwaited=TRUE  $\wedge$ 
  is_prep=FALSE  $\wedge$  is_ok=FALSE  $\wedge$  is_prep_pr=FALSE  $\wedge$  prep_state=prep_pr
  THEN state:=susp || susp_state:=sp || is_fatal  $\in$  BOOL END;
recoverPrepPr (refines recoverPrep) =
  WHEN state=susp  $\wedge$  susp_state=sp  $\wedge$ 
  is_ok=FALSE  $\wedge$  is_fatal=FALSE  $\wedge$  is_prep_pr=FALSE
  THEN state:=init || init_state:=prep || prep_state:=prep_pr ||
  cmd:=no_cmd || wwaited:=FALSE || is_ok:=FALSE ||
  idDataSent:=FALSE || respRead:=FALSE || respValid:=FALSE || userValid:=FALSE END;
nIdDS (refines failPrepPr) =
  WHEN state=init  $\wedge$  init_state=prep  $\wedge$  prep_state=prep_pr  $\wedge$  is_prep=FALSE
   $\wedge$  is_prep_pr=FALSE  $\wedge$  is_ok=FALSE  $\wedge$  wwaited=TRUE  $\wedge$  idDataSent=FALSE
  THEN state=susp || susp_state=sp || is_fatal  $\in$  BOOL END;
...
END

```

The specification presented on the listing above, although more concrete, is not yet implementable. Nonetheless, it provides very good understanding of what actions should be taken in order to ensure stability and fault tolerance.

## 7. Related Work

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 Ilić et al. [9]. 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.

Refinement patterns in the Event-B method were also investigated by Alexei Iliasov [8], but with respect to the rapid development of dependable systems. The author explores a method for mechanised transformation of formal models and merges theory with practice by implementing the tool that supports the formerly created patterns language. Since automation is less error-prone than manual coding, applying patterns with the use of the created tool is profitable for the dependable systems development. We also rely on patterns in order to prevent introducing the errors into the system development, making the construction of the system process more dependable. However, we are not concerned about creating a language for the patterns. Instead we benefit from the progress diagrams through the readability and the intuition they provide.

An approach relating formal and informal development is used in the research of Claudia Pons [17], where the formally defined refinement methodology is submerged into UML-based development. The method is described by the term “formal-to-informal”, treated as a complement of the “informal-to-formal” approach standing for translating the graphical notation into formal language. The presented methodology is based on the Object-Z formal language and UML structures. It presents an object decomposition pattern and a non-atomic operation refinement via examples of classes. In our research we focus on statemachines instead of classes and combine the formal and informal approaches, which in our case are complementary to each other. Moreover, we use Event-B in order to have a tool support for our development.

Defining standards in semantics for different level of abstractions in system level design has been studied by Junyu Peng, Samar Abdi, and Daniel Gajski in [15]. The authors’ approach to system development relies on the automation of the refinement process via tool support. The main focus in their research is to improve robustness and usefulness of the system design, even if the methodology aims at the architecture of the system in general. Their effort is towards rapid prototyping and evaluation of several design points, while our approach is of a formal nature, focusing on the correctness of the system created in a stepwise manner.

## 8. Conclusions and Future Work

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.

We have considered the possibility of developing tool support for drawing progress diagrams and automatically generating a new refinement from the progress diagram and the previous level model. The most likely route for tool support is to extend the UML-B tool [19], which is already an extension of the Event-B tool set. The complete tool set is based on the Eclipse development environment as a 'rich client platform'. UML-B provides a graphical drawing tool for drawing state machine diagrams (as well as class diagrams) and converting them automatically into Event-B where the Event-B static checker and prover automatically perform verification on the model. UML-B uses the 'Eclipse Modelling Framework' (EMF) to generate a repository for UML-B models from a meta-model diagram. The UML-B meta-model defines the abstract syntax of the UML-B language. The drawing tool is based on the 'Graphical Modelling Framework' (GMF). We envisage a new meta-model for progress diagrams, that extends the UML-B meta-model to define the refinement relations, that we have described in this paper, mapping individual elements of an existing UML-B model to newly created UML-B elements. The diagrammatic part of the progress diagram editor would consist of a reduced UML-B Statemachine diagram. The tabular part of the progress diagram is an elegant view of the refinement properties but it is not the most suitable interface for editing them considering that they are based on the existing UML-B meta-classes. Therefore, a new editor interface (diagrammatic, tree structured or tabular) will be developed for defining the refinement properties as extensions to the referenced existing model elements. The tabular part of the progress diagram will be automatically generated as a read-only view of the refinement properties. A builder will be provided to generate the new refined UML-B model based on the progress diagram refinement model. Since the generated model is a UML-B model, the existing tools will automatically generate an equivalent Event-B model as soon as it is created.

### **Acknowledgements**

We would like to thank Dr Linas Laibinis and Dr Dubravka Ilić 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. A. Iliashov. Refinement Patterns for Rapid Development of Dependable Systems. *Proc of Engineering Fault Tolerant Systems Workshop* (at ESEC/FSE, Dubrovnik, Croatia), ACM Digital Library, 4 September, 2007.
9. D. Ilić and E. Troubitsyna. A Formal Model Driven Approach to Requirements Engineering. TUCS Technical Report No 667, Åbo Akademi University, Finland, February 2005.
10. S.M. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337-356, April 1993.
11. 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)
12. Object Management Group. *Unified Modelling Language Specification - Complete UML 1.4 specification*, September 2001.
13. 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)
14. M. Olszewski and M. Płaska. *Memento system*. <http://memento.unforgiven.pl>, 2006.
15. J. Peng, S. Abdi and D. Gajski. Automatic Model Refinement for Fast Architecture Exploration. In *Proc of the 15th International Conference on VLSI Design (VLSID.02)*.2002, p. 332, IEEE Computer Society .
16. M. Płaska, M. Waldén and C. Snook: Documenting the Progress of the System Development. In *Proc. of Workshop on Methods, Models and Tools for Fault Tolerance*, Oxford, UK, July 2007
17. C. Pons. Heuristics on the Definition of UML Refinement Patterns. In *SOFSEM 2006: Theory and Practice of Computer Science*, LNCS 3831, pp. 461-470, 2006. Springer.
18. C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92-122, January 2006. ACM Press.
19. C. Snook and M. Butler. UML-B and Event-B: an integration of languages and tools. In *Proc. of The IASTED International Conference on Software Engineering – SE2008*, Innsbruck, Austria, February 2008.
20. 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.
21. E. Troubitsyna. *Stepwise Development of Dependable Systems*. Turku Centre for Computer Science, TUCS, Ph.D. thesis No. 29. June 2000.
22. 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.