Pontus Boström | Mats Neovius | Ian Oliver | Marina Waldén

Formal Transformation of Platform Independent Models into Platform Specific Models in MDA

Turku Centre *for* Computer Science

TUCS Technical Report
No 759, March 2006

Formal Transformation of Platform Independent Models into Platform Specific Models in MDA

Pontus Boström
Åbo Akademi University, Department of Information Technologies
Turku Centre for Computer Science (TUCS)
Lemminkäisenkatu 14 A, 20520 Turku, Finland
Pontus.Bostrom@abo.fi

Mats Neovius
Åbo Akademi University, Department of Information Technologies
Lemminkäisenkatu 14 A, 20520 Turku, Finland
Mats.Neovius@abo.fi

Ian Oliver
Nokia Research Center
Itämerenkatu 11-13, 00180 Helsinki, Finland
Ian.Oliver@nokia.com

Marina Waldén
Åbo Akademi University, Department of Information Technologies
Turku Centre for Computer Science (TUCS)
Lemminkäisenkatu 14 A, 20520 Turku, Finland
Marina.Walden@abo.fi

TUCS Technical Report
No 759, March 2006

Abstract

This paper introduces a method for formal transformation of platform independent models (PIM) to platform specific models (PSM) in a model driven architecture (MDA) context[1]. The models are constructed using statemachines in the Unified Modelling Language (UML). As a formal framework for reasoning about the models we use Event B. In this paper we focus on fault tolerance features. Fault tolerance is not considered in the PIM in order to make the models reusable for different platforms. On the other hand, the PSM often has to consider platform specific faults. However, fault tolerance mechanisms cannot usually be introduced as a refinement in the PSM. In this paper we introduce a model transformation of the PIM in order to preserve refinement properties in the development of the PSM. Design patterns are used for guiding the development. UML is widely used in industry and therefore this development method can be beneficial for developing reliable applications in many different application areas.

Keywords: Model Driven Architecture, UML, Statemachines, Event B, Refinement

TUCS Laboratory
Distributed Systems Design Laboratory

[1]Work done within the RODIN-project, IST-511599

1 Introduction

A platform independent model (PIM) in a Model Driven Architecture (MDA) [13] context considers only features in the problem domain. In order to implement the platform independent model, the model is transformed into a platform specific model (PSM) that takes into account implementation issues for the platform where the system will run. The PSM is not necessarily a refinement of the PIM, since it can introduce features that are not considered there at all [15]. For example, fault tolerance and other platform specific features should not be included in the PIM, since every possible platform where the system could run would have to be taken into account. All potential platforms might not even be known at the time the PIM is created [4, 11].

We use UML [14] to describe the platform independent and platform specific models. Here we concentrate on statemachines [7]; we do not consider the object oriented features of UML. To have a formal semantics and good tool support for analysis, the statemachines are translated to Event B. Event B [3, 12] is a formalism based on Action Systems [5] and the B Method [2] for reasoning about distributed and reactive systems. It supports stepwise refinement of specifications and it is also compatible with UML statemachines [16].

In order to anticipate all the different restrictions that will be encountered on a specific platform, the fault handling mechanisms and other platform specific features in the PIM would have to be very general. Hence, they would not provide any useful information and could restrict future transformations to other platforms. We introduce an automatic transformation of the PIM to allow a very abstract definition of fault tolerance and other platform specific features. These platform specific features can then be refined to concrete features in the platform specific model. We give extra rules for ensuring deadlock freeness and preserving the behaviour of the PIM. In this paper we focus on fault tolerance features and we use pattern to facilitate the introduction.

Section 2 gives a short presentation of UML and Section 3 describes Event B. The translation of UML statemachines to Event B is then given in Section 4. The PIM to PSM transformation is introduced in Sections 5 and 6. Patterns for fault tolerance are presented in Section 7 and in Section 8 we conclude.

2 UML

An application developed using object oriented methodology [1, 17] consists of a set of objects that communicate by sending messages. Each object consists of a set of variables and a set of operations describing the functionality of the object. Messages are assumed to be operation calls. Furthermore, objects run concurrently and the communication between them is instantaneous. The behaviour of the objects is described by statemachines [7]. Here

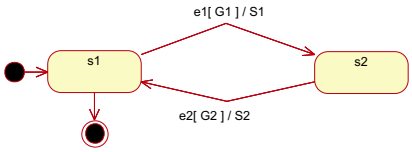


Figure 1: A simple statemachine

we assume that at most one statemachine is used for each object.

The statemachine specification in UML is large and complex [14]. For simplicity we will only use a subset of it. We consider only statemachines containing normal, initial and final states and no composite states. We assume that all statemachines containing composite states have been flattened.

Transitions between states model the execution of the system. They can be labelled by events and each event corresponds to an operation. Hence, statemachines in different objects can communicate by sending events (operation calls). In UML, events that cannot fire a transition are normally implicitly consumed. However, we assume that events are always deferred until they can fire a transition. If a transition is without an event, it can be non-deterministically fired by an arbitrary event. Transitions can have *guards* and *actions* that take into account other variables than the state in the enclosing object. A guard is a predicate that has to evaluate to *true* before the transition can be fired. An action is a substitution that is executed when the corresponding transition is fired.

A simple statemachine is shown in Figure 1. The statemachine contains two states s_1 and s_2. There are two transitions E_1 and E_2 between the states. Transition E_1 is triggered by event e_1 and it has the guard G_1 and the action S_1.

The specification of statemachines in UML contains also other features than the ones described here, such as e.g., composite states, history states, activities, entry- or exit-actions inside states. However, the aim of this paper is not to give a complete formal semantics to UML statemachines, but to consider mapping a PIM into a fault tolerant PSM within the UML environment.

3 Event B

Event B [12] is a formalism based on Action Systems [5] and the B Method [2], and it is related to B Action Systems [21]. It has been developed for reasoning about distributed and reactive systems. An Event B specification consists of an abstract model that can be refined to a concrete model in a stepwise manner.

MODEL $\mathcal{M}$
SEES
$\mathcal{C}$
VARIABLES
v
INVARIANT
$I(s, c, v)$
INITIALISATION
$S_0(s, c, v)$
EVENTS
$E_1 \mathrel{\hat{=}}$
WHEN $G_1(s, c, v)$ **THEN** $S_1(s, c, v)$ **END** ;
$E_2 \mathrel{\hat{=}}$
WHEN $G_2(s, c, v)$ **THEN** $S_2(s, c, v)$ **END** ;
END

CONTEXT $\mathcal{C}$
SETS
s
CONSTANTS
c
PROPERTIES
$P(s, c)$
END

Figure 2: An abstract Event B model

3.1 Abstract model

An Event B model consists of variables giving the statespace and events for describing the behaviour of the system [12]. Consider model $\mathcal{M}$ in Figure 2. The context $\mathcal{C}$ of the model provides definitions of sets s and constants c, where $P(s, c)$ describes their properties. The set of variables in the model is given by v. Their types and properties are given in the invariant $I(s, c, v)$. The behaviour of the model is described by the events E_1 and E_2. Each event consists of a guard G_i and a substitution S_i. When the guard G_i evaluates to *true* the event E_i is said to be enabled and the substitution S_i can be executed. Enabled events are chosen non-deterministically for execution.

A number of proof obligations need to be discharged in order to show that an Event B model is consistent [12]. To generate the proof obligations every substitution $S_i(s, c, v)$ is translated to a before-after predicate $PS_i(s, c, v, v')$. For example, if the substitution is $S(s, c, v) \mathrel{\hat{=}} (v := F(s, c, v))$ then the before-after predicate is $PS(s, c, v, v') \mathrel{\hat{=}} (v' = F(s, c, v))$. The first two proof obligations concern the correctness of the initialisation.

Mod1: $P(s, c) \Rightarrow \exists v'.PS_0(s, c, v')$

Mod2: $P(s, c) \wedge PS_0(s, c, v') \Rightarrow I(s, c, v')$

The proof obligations state that the initialisation should be possible (Mod1) and it should establish the invariant (Mod2). Note that the before-after predicate for the initialisation only refer to the new value of the variables v. The correctness of each event E_i is ensured by the following proof obligations.

Mod3: $P(s, c) \wedge I(s, c, v) \wedge G_i(s, c, v) \Rightarrow \exists v'.PS_i(s, c, v, v')$

Mod4: $P(s, c) \wedge I(s, c, v) \wedge G_i(s, c, v) \wedge PS_i(s, c, v, v') \Rightarrow I(s, c, v')$

The first proof obligation states that the substitution in each event E_i should be possible (Mod3) and the second one that the event has to maintain the invariant (Mod4).

REFINEMENT $\mathcal{M}_1$
REFINES $\mathcal{M}$
SEES
$\mathcal{C}$
VARIABLES
w
INVARIANT
$J(s, c, v, w)$
INITIALISATION
$R_0(s, c, w)$
EVENTS
$E_1 \mathrel{\hat{=}}$
WHEN $H_1(s, c, w)$ **THEN** $R_1(s, c, w)$ **END** ;
$E_2 \mathrel{\hat{=}}$
WHEN $H_2(s, c, w)$ **THEN** $R_2(s, c, w)$ **END** ;
$F_1 \mathrel{\hat{=}}$
WHEN $N_1(s, c, w)$ **THEN** $T_1(s, c, w)$ **END** ;
END

Figure 3: An Event B refinement model

3.2 Refinement

An Event B model can be refined [12]. As an example we have model $\mathcal{M}_1$ in Figure 3 that is a refinement of the model $\mathcal{M}$. The variables in the refined model is given by w. The relation between the variables in the abstract model and the refined model is given by the refinement invariant $J(s, c, v, w)$. The guard G_i and substitution S_i in event E_i is refined by the guard H_i and substitution R_i, respectively. A new event F_1 may also be introduced. In order to show that event E_i is refined in a correct manner the following proof obligations need to be discharged.

Ref1: $P(s, c) \wedge I(s, c, v) \wedge J(s, c, v, w) \wedge H_i(s, c, w) \Rightarrow \exists w'.PR_i(s, c, w, w')$

Ref2: $P(s, c) \wedge I(s, c, v) \wedge J(s, c, v, w) \wedge H_i(s, c, w) \Rightarrow G_i(s, c, v)$

Ref3: $P(s, c) \wedge I(s, c, v) \wedge J(s, c, v, w) \wedge H_i(s, c, w) \wedge PR_i(s, c, w, w') \Rightarrow \exists v'.(PS_i(s, c, v, v') \wedge J(s, c, v', w'))$

The proof obligations states that the refined substitution R_i is possible (Ref1), the guard of the event is strengthed (Ref2) and that there is an assignment to the variables in the abstract model corresponding to the assignment in the refined substitution under relation J (Ref3). The proof obligations for new events F_i are similar to the ones above. However, they refine *skip* and, hence, the before-after predicate in the abstract specification is $PS_n(s, c, v, v') \mathrel{\hat{=}} (v' = v)$.

In order to ensure the correctness of the entire model, two additional proof obligations need to be discharged. The refined system cannot deadlock or terminate more often than the abstract one (Ref4).

Ref4: $P(s, c) \wedge I(s, c, v) \wedge J(s, c, v, w) \wedge G_i(s, c, v) \Rightarrow H_i(s, c, w) \vee N_1(s, c, w) \vee \ldots \vee N_n(s, c, w)$

If an event is enabled in the abstract model it is also enabled in the refined model or some new events are enabled. This is the strong version of the proof obligation for deadlock freeness. Finally, we need to show that the new events terminate when executed in isolation, since they are not allowed to take control forever. We assume that we have a variant $V(s, c, w)$ that is a well founded structure $(\mathbb{N}, \leq)$. Each new event then has to decrease the variant (Ref5).

$$\text{Ref5:} \quad \begin{array}{l} P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge N_i(s,c,w) \wedge PT_i(s,c,w,w') \Rightarrow \\ V(s,c,w') \in \mathbb{N} \wedge V(s,c,w') < V(s,c,w) \end{array}$$

These proof obligations are necessary and sufficient to show that an Event B model is consistent and that the refinement is correct.

A recently introduced feature in Event B is the, so called, anticipating events [3]. Sometimes a new event is needed in a refinement that also modifies old variables. The event is then added as an anticipating event in previous refinement steps. An anticipating event can perform any substitution that maintains the invariant. We still have to prove that the new events refining the anticipating events terminate when executed in isolation. Anticipating events is only syntactic sugar for actually introducing the events in earlier refinement steps and introducing extra variables for their variant.

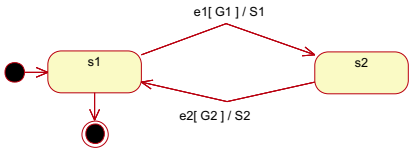
4 Translation of statemachines to Event B

Since UML is a specification language that is widely used in industry, we create behavioural models using UML statemachines. In order to be able to formally reason about the models, we translate them to Event B. There are several translations from UML to B [16, 19]. However, here we present a translation of a subset of UML, using a semantics concerning events and communication that is suitable for our purpose.

4.1 Transitions and Events

The statemachine in Figure 1 is translated to Event B as shown in Figure 4. The states of the statemachine are given as an enumerated set $S = \{s_1, s_2, exit\}$ in Event B. The current state of the statemachine is modelled as a variable $s \in S$. The initial state in a UML statemachine gives the initial value of the state variable s. The exit states are modelled as a single state, $exit \in S$. The variables v in $\mathcal{M}$ are the variables of the UML statemachine. The transitions correspond to events in Event B. The events E_1 and E_2 in Event B (later called B events) corresponds to the transitions triggered by the UML events e_1 and e_2, respectively. The guards G_i and substitutions S_i refer to the variables v.

The proof obligations for transitions are the standard proof obligations for events in Event B. However, it is a desirable property that a statemachine should not deadlock. The only state where no transition should be enabled is



MODEL $\mathcal{M}$
SETS
$S = \{s_1, s_2, exit\}$
VARIABLES
v, s
INVARIANT
$I(v, s)$
INITIALISATION
$S_0(v) \parallel s := s_1$
EVENTS
$E_1 \mathrel{\hat{=}}$
WHEN $s = s_1 \wedge G_1$
THEN $S_1 \parallel s := s_2$
END
$E_2 \mathrel{\hat{=}}$
WHEN $s = s_2 \wedge G_2$
THEN $S_2 \parallel s := s_1$
END
$E_{exit} \mathrel{\hat{=}}$
WHEN $s = s_1$ **THEN** $s := exit$ **END**
END

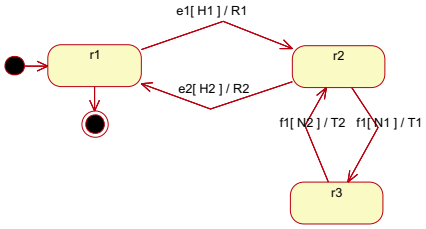
Figure 4: Translation of a UML statemachine $\mathcal{M}$ to Event B

in state *exit*. To ensure that this holds we introduce an extra proof obligation (Exit1).

Exit1: $I(v, s) \wedge \neg((s = s_1 \wedge G_1) \vee \ldots \vee (s = s_n \wedge G_m)) \Rightarrow s = exit$

The abstract model in Figure 4 can be refined to take into account more features and to make the model implementable. In figure 5 we refine the abstract variables v with concrete variables w. The guards and actions in the transitions of the abtract model are refined to H_i and R_i, respectively, to take the concrete variables w into consideration. The state variable s is refined by r. The relation between s and r is given as $J_S(s, r) \mathrel{\hat{=}} (s = s_1 \Leftrightarrow r = r_1) \wedge (s = s_2 \Leftrightarrow r \in \{r_2, r_3\})$. Two new transitions with UML event f_1 have also been introduced in the model (transitions F_{11} and F_{12} in Event B). Translation of the refined statemachine $\mathcal{M}_1$ is performed in the same manner as for the abstract model.

A UML event can be considered to be an operation in an object as described in Section 2. Thus, we can consider each UML event to be the non-deterministic choice of its transitions. Assume UML event e consists of transitions $E_1, \ldots, E_n$. Then the behaviour in Event B for e is given as $e \mathrel{\hat{=}} E_1 [\!] E_2 [\!] \ldots [\!] E_n$. Transition without an UML event are included in the behaviour of e, since these transitions can be triggered by any UML event. The behaviour of e should be refined when the statemachine is refined. This interpretation of UML events gives rules for labeling transitions with UML events: A transition without an UML event can be refined to a transition with any UML event. The UML event cannot be changed on a transition during the refinement process. Furthermore, if a UML event e can trigger a transition in the abstract statemachine it should also be possible in the refined statemachine. Since we have deferred UML events, this can be



REFINEMENT $\mathcal{M}_1$
REFINES $\mathcal{M}$
SETS
$R = \{r_1, r_2, r_3, exit\}$
VARIABLES
w, r
INVARIANT
$J(v, w, r) \wedge J_S(s, r)$
VARIANT
$V(w, r)$
INITIALISATION
$R_0(w) \parallel r := r_1$
EVENTS
$E_1 \mathrel{\widehat{=}}$
WHEN $r = r_1 \wedge H_1$
THEN $R_1 \parallel r := r_2$
END
$E_2 \mathrel{\widehat{=}} \ldots$
$F_{11} \mathrel{\widehat{=}}$
WHEN $r = r_2 \wedge N_1$
THEN $T_1 \parallel r := r_3$
END
$F_{12} \mathrel{\widehat{=}} \ldots$
$E_{exit} \mathrel{\widehat{=}}$
WHEN $r = r_1$
THEN $r := exit$
END
END

Figure 5: Translation of a refined UML statemachine $\mathcal{M}_1$

proved by showing the strong version of deadlock freeness in combination with termination of new transitions.

4.2 System consisting of several objects

Applications usually consist of more than one object. Consider an application consisting of the objects $\mathcal{A}$ and $\mathcal{B}$ where the behaviour of the objects is described using UML statemachines. Object $\mathcal{A}$ has the variables v_A, the operations $e_1, \ldots, e_m$ and the states S_A, while the object $\mathcal{B}$ has the variables v_B, the operations $f_1, \ldots, f_n$ and the states S_B.

In order describe the composition of objects $\mathcal{A}$ and $\mathcal{B}$ we rely on the parallel composition of action systems with procedures [18]. The parallel composition, $\mathcal{A} \parallel \mathcal{B}$, of the statemachines describing the behaviour of objects $\mathcal{A}$ and $\mathcal{B}$ can be defined in Event B. We assume that the sets of variables v_A and v_B are disjoint, otherwise the variables are renamed before composition. In the composed Event B model the variables are merged, $v_A \cup v_B$. The state in the composed statemachine is $S_A \times S_B$. A transition from s_i to s_j of the form $e_i[G_i]/S_i$ in $\mathcal{A}$ is translated to the following B event:

$E_i \mathrel{\widehat{=}}$
WHEN $s_A = s_i \wedge G_i$
THEN $S_i \parallel s_A := s_j$
END

We note that the state of $\mathcal{B}$ is not changed here.

Communication between objects is performed by sending events, which corresponds to operation calls. A transition $e_j[G_j]/S_j \parallel f_k$ in $\mathcal{A}$ that sends an event f_k, triggering transition $f_k[H_k]/R_k$ in object $\mathcal{B}$, models instantaneous communication between the objects. We assume that each statemachine involved in the communication will only make one transition. Hence, A transition in $\mathcal{B}$, labelled f_k, cannot send an event back to $\mathcal{A}$. The two transitions are then translated together into Event B as follows:

$E_j \mathrel{\widehat{=}}$
WHEN $(s_A = s_j \wedge G_j) \wedge (s_B = s_k \wedge H_k)$
THEN $S_j \parallel s_A := s_i \parallel R_k \parallel s_B := s_l$
END

The communication can take place when both the guard of the transition in $\mathcal{A}$ and $\mathcal{B}$ are enabled. The actions of the transitions in $\mathcal{A}$ and $\mathcal{B}$ are then executed as one atomic operation. Note that if f_k triggers several transitions there needs to be a composed transition for each of them.

To prove that the composition $\mathcal{A}' \parallel \mathcal{B}'$ is a refinement of $\mathcal{A} \parallel \mathcal{B}$ we only need to show that $\mathcal{A} \sqsubseteq \mathcal{A}'$ and $\mathcal{B} \sqsubseteq \mathcal{B}'$ [18]. Hence, the proof obligations in Section 3 should be discharged for both the objects $\mathcal{A}$ and $\mathcal{B}$, in order to prove $\mathcal{A} \parallel \mathcal{B} \sqsubseteq \mathcal{A}' \parallel \mathcal{B}'$.

5 Introduction of platform specific features in UML

In order to implement a PIM we need to transform it into a PSM. The PSM is not necessarily a refinement of the PIM, since the PIM does not consider platform specific features. Here we are mainly interested in introduction of fault tolerance features; we can even consider fault tolerance to be a platform in its own right (cf. [4, 11]). We can make the following observations: the behaviour in the PIM should be the "normal" behaviour of the PSM and the new behaviour in the PSM relates mainly to tolerance of faults that can occur on a specific platform.

The faults can be divided into three groups: the first group consists of faults where the only remedy is to terminate the application. The second group consists of faults that cannot be recovered from when the mechanisms for fault tolerance are introduced as a refinement of the PIM. An example of such a fault could be inserting an item into a buffer. The PIM considered the buffer to have infinite length, while in the PSM it has a finite length. When the buffer gets full, messages are dropped, but the application can otherwise continue its operation. The last group of faults consists of faults that the fault tolerance mechanisms always can recover from and where the fault tolerance mechanism can be introduced as a refinement. This group of faults are not a problem, since the PSM can still be a refinement of the PIM in this case.

We would like to preserve as many refinement properties as possible in the transformation from PIM to PSM. The following properties of the PIM are required to be preserved in the PSM:

1. The sequence of valid calls to public operations are maintained in the PSM or the statemachine has reached state *exit*.

2. New public operations (UML events) are not introduced. Hence, an object does not require new interactions from its environment.

3. New behaviour violating the refinement relation between the PIM and the PSM cannot take control forever.

4. There should be a trace in the PIM that is also a possible trace in the PSM. Hence, it should be possible to execute the PSM using only the transitions in the PIM.

The rules above can also be expressed as restrictions on the statemachine in the PSM. In the view of the environment, a UML statemachine accepts a language over an alphabet consisting of the events. Assume that statemachine in the platform independent model accepts the language L. The alphabet of the language is the public operations of the object. Consider two strings L_1 and L_2 in L such that $L = L_1L_2$. In order to introduce fault tolerance, we can add new error handling mechanisms. Assume the error handling mechanism is represented by the string of events f and the error can occur between L_1 and L_2. The statemachine of the platform specific model can then accept the following language $L_1L_2 + L_1fL_2 + L_1f$. This means that the statemachine operates either normally, recovers and continues with its normal operation or it terminates.

6 Transforming a PIM into a PSM

In order to transform the PIM into a PSM we use design patterns. Design patterns are template solutions for solving commonly occurring problems. Fault tolerance patterns will be discussed in more detail in Section 7.

6.1 Introducing anticipating events

To enable transformation of the PIM to a PSM we use anticipating events in Event B. We transform the PIM to a model having all the possible anticipating events modelling very abstract fault tolerance features. Figure 6 illustrates how a platform independent model $\mathcal{M}$ is transformed into a platform specific model $\mathcal{M}''$. First $\mathcal{M}$ is automatically translated to a model $T(\mathcal{M})$ including all the possible anticipating events. The model $T(\mathcal{M})$ is hidden from the developer of the PIM. He/She will only have to consider the models $\mathcal{M}$, $\mathcal{M}'$ and $\mathcal{M}''$. To obtain a model $\mathcal{M}'$ with platform specific features, a pattern p_1 is applied to the PIM $\mathcal{M}$ by the developer. This procedure can be repeated until all platform specific features have been introduced. The result obtained is a model that have similar functionality as $\mathcal{M}$, but can have several platform specific features, e.g. fault tolerance.

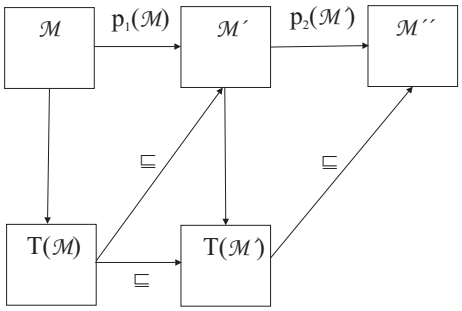


Figure 6: Transforming a platform independent model $\mathcal{M}$ into a platform specific model $\mathcal{M}''$

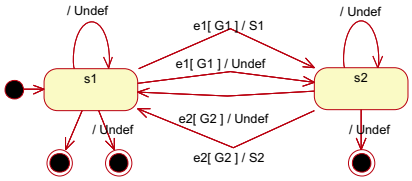


Figure 7: The transformation of the platform independent model $T(\mathcal{M})$ in Figure 1

The obtained model $\mathcal{M}'$ is a refinement of $T(\mathcal{M})$, but not necessarily of the platform independent model $\mathcal{M}$.

The problem of fault tolerance can be divided into two separate parts, the modelling of the occurrence of an error and the error handling. To model the occurrence of errors we introduce one new transition for each transition in the platform independent model. These anticipating transition $e_i[G_i]/Undef$ have the same source, destination and guard as their corresponding transitions. They contain the action $Undef$, modelling that the execution of the action failed. To model error handling we introduce two extra transitions $/Undef$ for each state in the platform independent model. These anticipating transitions have no event and can, hence, be executed at any time. One transition models errors that the system can recover from, the other models termination of the system in the state *exit*. All the anticipating transitions can be refined to any behaviour, as for example fault tolerance.

An abstract platform independent model is presented in Figure 1 in Section 2. The model contains two states, s_1 and s_2 and two transitions between them. The corresponding transformed PIM $T(\mathcal{M})$ is shown in Figure 7, where anticipating transitions are introduced for both the states s_1 and s_2, as well as between these two states.

6.2 Validation of the platform specific model

We transform the PIM $\mathcal{M}$ to a PSM $\mathcal{M}'$ using a pattern p that takes into account the transformation rules in this paper. The model $\mathcal{M}'$ is shown within Figure 8. Validation of the PSM is performed within the Event B framework, where we can show that the PSM $\mathcal{M}'$ is a refinement of the transformed platform independent model $T(\mathcal{M})$. In Section 5 we gave four additional properties that the PSM $\mathcal{M}'$ should satisfy with respect to the original PIM $\mathcal{M}$. They are related to proof obligations in Event B as follows. The sequence of valid calls (1) is preserved, since we have deferred events and we prove strong deadlock freeness for each transition in Event B. We can check syntactically on the UML model that no new public events are introduced (2). New behaviour is not allowed to take control forever (3), which is guaranteed with the proof obligations for anticipating transitions and new transitions in Event B. Condition (4) stating that there is behaviour common to the PIM and PSM requires extra proof obligations in Event B.

The fourth requirement in Section 5 is needed, since we introduce several extra transitions in $T(\mathcal{M})$ compared to $\mathcal{M}$. It is necessary to show that the behaviour of the PIM is still feasible in the PSM. First, we check that the initialisation $R_0(w, r)$ of the PSM $\mathcal{M}'$ can enable an already existing transition in the PIM or that it moves to state *exit* (PSM1).

PSM1: $\exists w', r'.(PR_0(w', r') \wedge (H_1(w', r') \vee \ldots \vee H_n(w', r') \vee r' = exit))$

Here every H_i denotes the Event B guard of a transition in the PSM that refines a transition in the PIM. Furthermore, for every transition in the PSM **WHEN** $H_j(w, r)$ **THEN** $R_j(w, r)$ **END** that is a refinement of a transition in the PIM there should be assignment w', r' to the variables that enables a transition with translated guard H_i or leads to the state *exit* (PSM2).

PSM2: $I(v, s) \wedge J(v, w, s, r) \wedge H_j(w, r) \Rightarrow$
$\exists w', r'.(PR_j(w, w', r, r') \wedge (H_1(w', r') \vee \ldots \vee H_n(w', r') \vee r' = exit))$

When these proof obligations hold the statemachine is not allowed to use only fault tolerance mechanisms. Note that the proof obligations are the same as the feasibility proof obligation with the the condition $(H_1(w', r') \vee \ldots \vee H_n(w', r') \vee r' = exit)$ added.

In the PIM we show that when all events in the system are disabled then all objects are in state *exit*. Hence, the PIM of the system does not deadlock. When we transform the PIM, we allow the introduction of new transitions to state *exit*. Even if the statemachine in the PIM always terminates in state *exit,* this property does not necessarily hold in the PSM. We need an extra proof obligation to ensure that the statemachine only terminates in this state (Exit2).

Exit2: $I(v, s) \wedge J(v, w, s, r) \wedge \neg(H_1(w, r) \vee \ldots \vee H_n(w, r)) \Rightarrow (r = exit)$

As a final issue, we need to consider UML events on transitions. Several of the transitions in $T(\mathcal{M})$ were not labelled with UML events. Since UML events are operations in objects, there should not be any transitions without events in the final PSM. However, new private UML events are introduced that either trigger new transitions or transitions refining the anticipating transitions. These extra rules ensure that the desired properties given in Section 5 holds.

7 Fault Tolerance Patterns

A pattern introducing new functionality is dependent upon the original model and must satisfy the system constraints. In this case it means that the fault tolerance pattern must not change the original functionality of the system. This implies that the fault tolerance features have to be refined during the development. Thus, for all trace(s) in the original $\mathcal{M}$, there must be corresponding valid trace(s), in the successively developed systems, including the transition containing $Undef$. Consequently, after fault tolerance has been introduced, it can be seen as a "slave" system evolving in parallel. Introducing the "slave" system early provides a higher level of abstraction, but makes the refinement more complicated in contradiction to initiating it later.

In this paper we add fault tolerance as a final step of the development. Considering Figure 8, abstract fault tolerance features are added to the model $T(\mathcal{M})$. The added features include transitions containing $Undef$ that can be refined to anything satisfying the abstract invariant. The transformation that is performed automatically follows a pattern for adding the anticipating events and hence, model $T(\mathcal{M})$ is not a refinement of $\mathcal{M}$. When introducing fault tolerance according to a pattern p into the PSM $\mathcal{M}'$, we will have the case that $\mathcal{M}'$ is a refinement of $T(\mathcal{M})$.

In general, a fault is a defect in the system that possibly can manifest itself as an error, which might result in a system failure [20]. Because of the unpredictability of the environment, faulty behaviour is evident. Fault tolerance refers to a method for designing a system so that it is capable of operating, possibly at a reduced level, rather than failing completely when some part of the system fails. Consequently, a fault tolerant system is capable of handling unexpected erroneous events in a sensible manner. A fault tolerance procedure always starts with error detection, followed by a diagnosis and the outcome of the error handling method. In this paper the detection is modelled as a transition containing $Undef$, while the diagnosis is completely dependent on the uniqueness of the current state. The outcome of the fault tolerance is achieved through applying a handling method designed for errors occurring in that specific state. Consequently, an error in a certain state is always tackled according to the same pattern.

The PSM model can handle the errors in three distinct ways. Optimally a recovery from the error is performed; otherwise the system prefers a

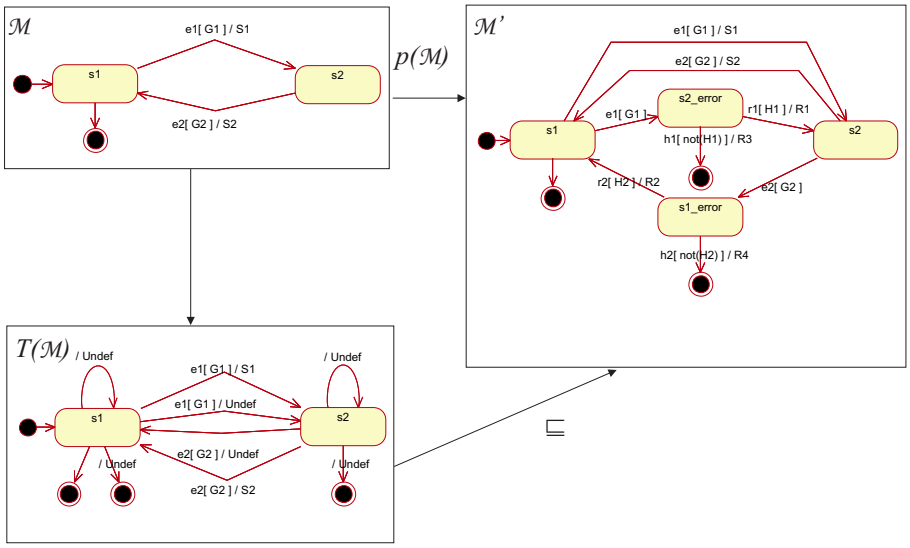


Figure 8: Transformation of the PIM $\mathcal{M}$ into a PSM $\mathcal{M}'$

shutdown in a clean manner, i.e., a software reboot under controlled circumstances conducting damage avoidance procedures. If neither can be chosen, the system will suffer an uncontrolled "total" failure.

In our transformation from PIM to PSM we prohibit usage of history states and of traces. Hence, if there is more than one possible path leading to the present state, the system cannot know its prior events. Consequently, any pattern preserving the system conditions is deduced from a unique situation of the system, making checkpoint dependent recovery unattainable [8]. These constraints demand more of the development of fault tolerance.

When developing a fault tolerant system, according to the method in this paper, it is deadlock free and preserves the invariant. The patterns are bounded within the strict constraints proposed here. This means that the system $T(M)$ (in Figure 8) will have the following structure, where L_i stands for a subset of the language (UML events) accepted by a statemachine in the PIM, f for an erroneous event and* denotes the Kleene star: $L_1 f^* L_2 f^* \ldots L_n f^*$. Hence, any number of faults can occur between the events of the PIM. If fault tolerance would evolve in parallel with the model, every added feature can potentially change the desired recovery method.

In the PSM we can refine the error f in $T(\mathcal{M})$ (transition with $Undef$ in Figure 8) to a combination of handled crash h and error recovery r in

$\mathcal{M}'$. The expression $L_1L_2 + L_1rL_2 + L_1h$ in $\mathcal{M}'$, gives the refined fault tolerance, where + is disjunction. Every error state would have to contain at least the worst case scenario of unexpected total breakdown, leading to hard reboot. However, the unexpected total breakdown is out of the scope of this paper and is not modelled here. Our expression models normal behaviour (L_1L_2), successful recovery (L_1rL_2) and handled crash after possible recovery attempts (L_1h). By following this method we can introduce fault tolerance in the PSM while preserving the main functionality of the PIM.

8 Conclusions

In this paper we presented a method for transforming a PIM into a PSM in a MDA context. Our transformation rules are mainly aimed at introducing fault tolerance features into the models. We consider behavioural models constructed using UML statemachines and we use Event B as the underlying formal framework. A PIM does not consider platform specific features such as fault tolerance. However, the PSM often has to consider platform specific faults, but the fault tolerance mechanisms cannot necessarily be introduced as a refinement of the PIM. The transformation rules in this paper will ensure that certain desirable properties are preserved in the PSM.

Adding features to a model that do not obey refinement rules has been investigated before. Retrenchement [6] is an approach to make exceptions to refinement rules in a structured manner. For our purposes retrenchement is unnecessarily complicated and refinement, as well as, anticipating events [3] is sufficient to introduce platform specific features in the PSM.

Fault tolerance is often considered directly in the abstract specification (PIM). Adding fault tolerance in B has been investigated before by Laibinis and Troubitsyna in e.g. [9, 10]. However, we like to construct the PIM without considering fault tolerance, in order to focus on the desired functionality and to make the models more reusable.

As future work, we aim at to applying the method on a case study to investigate its practicality and to develop reusable patterns. The method is not limited to only the small subset of UML statemachines given in the paper, but it can be extended to consider more features from the UML standard. UML is well known in industry and therefore this type of transformation rules can be beneficial in many application areas.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1998

[2] J. R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press. 1996

[3] J. R. Abrial, D. Cansell and D. Mï¿½y. Refinement and Reachability in Event B. In *Proceedings of the 4th International Conference of B and*

Z users - ZB2005: Formal specification and Development in Z and B, Guildford, UK, LNCS 3455, Springer, pp. 222-241, 2005

[4] C. Atkinson and T. Kuhne. A Generalised Notion of Platforms for Model-Driven Development. In: Sami Beydeda, Mattias Book and Volker Gruhn (eds.) *Model-Driven Software Development*. Springer, pp. 119-136, 2005

[5] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing*, pp. 131-142, 1983

[6] R. Banach and M. Poppleton. Retrenchment: an Engineering Variation on Refinement. In *Proceedings of FM-99*, LNCS 1709, Springer, 1999

[7] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, Elsevier Science Publishers, pp. 231-274, 1987

[8] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. IEEE Computer Society Press, pp. 1150 - 1158, 1986, ISBN: 0-8186-4743-4

[9] L. Laibinis and E. Troubitsyna. Fault Tolerance in a Layered Architecture: A General Specification Pattern in B. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM 2004)*, IEEE Computer Society, 2004

[10] L. Laibinis and E. Troubitsyna. Refinement of Fault Tolerant Control Systems in B. In *Computer Safety, Reliability and Security - Proceedings of SAFECOMP 2004*, LNCS, 3219, Springer, pp. 254-268, 2004

[11] T. Margaria and B. Steffen. Aggressive Model-Driven Development: Synthesising Systems from Models viewed as Constraints. *The Monterey Workshop Series 2003 Theme: Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation*, Chicago, Illinois, September, 2003

[12] C. Métayer, J. -R. Abrial and L. Voisin. Event-B Language, *Rodin Deliverable D7*. RODIN project, IST-511599, 2005 (accessed 23.02.2006) `http://rodin.cs.ncl.ac.uk/deliverables.htm`

[13] Object Management Group. Model Driven Architecture (MDA). (accessed 23.02.2006) `http://www.omg.org/mda/`

[14] Object Management Group. Unified Modeling Language (UML). (accessed 23.02.2006) `http://www.uml.org/`

[15] I. Oliver. Model Based Testing and Refinement in MDA Based Development. In: *Pierre Boulet (ed.) Advances in Design and Specification Languages for SoCs*. The ChDL Series, Springer, 0-387-26149-4, 2005

[16] E. Sekerinski and R. Zurob. Translating Statecharts to B. In *Proceedings of the third international Conference on Integrated Formal Methods, IFM 2002*, Turku, Finland, May 2002, LNCS 2335, Springer, 2002

[17] B. Selic, G. Gullekson and P. T. Ward. *Real-Time Object-Oriented Modelling*. John Wiley & Sons. 1994

[18] K. Sere and M. Waldén. Data Refinement of Remote Procedures. *Formal Aspects of Computing*, 12, pp. 278-297, 2000

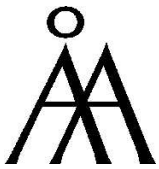
[19] C. Snook and M. Waldén. Use of U2B for specifying B action systems. In *Proceedings of RCS'02- International workshop on refinement of critical systems: methods, tools and experience*. Grenoble, France, 2002

[20] N. Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996

[21] M. Waldén and K. Sere. Reasoning About Action Systems Using the B-Method. *Formal Methods in Systems Design*, 13:5-35, 1998

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics



Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 952-12-1703-0
ISSN 1239-1891