Refinement Patterns for Fault Tolerant Systems

Alexei Iliasov
Newcastle University
Newcastle Upon Tyne, England
alexei.iliasov@ncl.ac.uk

Alexander Romanovsky
Newcastle University
Newcastle Upon Tyne, England
alexander.romanovsky@ncl.ac.uk

Abstract

The paper puts forward the idea of using fault tolerance refinement patterns to assist system developers in disciplined application of software fault tolerance mechanisms in rigorous system design. Two patterns are proposed to support a correct introduction of recovery blocks and N-version programming into a system model; these are formally defined and their correctness proven. We also discuss several important issues involved in the use of these patterns in engineering systems, including tool support and pattern composition.

1 Introduction

It is well-known that, while designing system fault tolerance means, developers tend to make numerous mistakes. First of all, these involve misuse of fault tolerance mechanisms (see, for example, [4, 3]). There are many reasons for this, including the complexity of the system and its environment, as well as that of the abnormal system behaviour in general and recovery in particular. The existing solution to these problems is to offer a fault tolerance mechanism as a set of design abstractions supported by the required middleware services (packaged as a library with a well-defined API). Sometimes this solution is backed by design patterns or aspects to help developers avoid mistakes in using it (e.g. [12]). The problem with this is that this approach does not target earlier development phases, creating a dangerous gap between system requirements and its implementation.

Formal methods have proven to be successful in developing a number of critical systems (e.g. in transport, telecommunication and automotive industry). They are typically used for system specification and validation to help in fault avoidance or removal. Recently, there has been some research carried out on modelling fault tolerance at the earlier phases of system development (e.g. [11], [5]). A way of applying formal methods to achieve fault tolerance, rigorous design of the latter is now becoming an area of active research, prompted a growing understanding in the industry of the need to deal with earlier phases (e.g. architecture design) as a major means for improving the quality of products.

Formal methods are not, however, a panacea, with the main difficulties in applying them being complexity of use and scalability. This is why considerable efforts are now devoted to tool support, as exemplified by the ICT RODIN project, which has developed an open-source extendable Eclipse platform supporting B development [16]). However, even with powerful tool support, formal methods will not be fully accepted as a mainstream software engineering paradigm. The approach we have been working on, called refinement patterns [8], helps developers in applying formal methods using computer-aided model transformations as part of rigorous stepwise system development in B. These transformations are used to capture standardised development steps that rigorously introduce well-defined fault tolerance into the system. When these patterns are formally described, their correctness can be verified to ensure that model transformations preserve model correctness. Patterns can be applied and undone instantaneously in the course of modelling; they significantly reduce the number of proofs that need to be done to demonstrate model correctness.

We believe that once a large number of patterns has been accumulated, the automated model transformation supported by patterns will have a profound effect on formal modelling. With pattern instantiation being little more than a mouse click, a well-designed pattern library could do for formal modelling what class libraries have done for the mainstream system development using programming languages.

This paper focuses on fault tolerance refinement patterns, which would help system developers to apply software fault tolerance mechanisms in system design in a disciplined fashion. In reporting our work on developing two fault tolerance patterns, we are fully aware that further effort needs to be made to build a useful library of patterns that would support a wide variety of error detection, error recovery and fault handling mechanisms. Our first pattern

models the recovery block scheme, whereas the second one helps in introducing N-version programming into a model.

The outline of the paper is as follows. Firstly, we formally define the two patterns and prove their correctness (which, as we have found, is not a trivial task). After that we briefly discuss several issues involved in engineering fault tolerant systems with patterns: tool support, pattern composition, pattern libraries, pattern identification and design diversity.

2 Background

2.1 Event-B

The Event-B method [16, 14] has been recently created on the basis of the B formal method [1] to enhance developers' understanding of and reasoning about systems, including reactive and concurrent ones. In its core is a modelling method based on the concept of refinement. An Event-B specification is made of *events* which update a system state and *variables* which represent it. *Invariant* is used to express the desired system properties. Event-B development is a tree of specifications linked by refinement relations. The general form of an Event-B specification is as follows:

machine $m0$
variables v
invariant I
initialisation $RI(v', v)$
events
  ...
  $evt_i =$ **any** p_i
    where
      $G_i(p_i, v)$
    then
      $R_i(p_i, v', v)$
    end
  ...
end

refinement $m1$
refines $m0$
variables w
invariant J
initialisation $SI(w', w)$
events
  ...
  evt_j **ref** $evt_i =$ **any** p_j
    where
      $H_j(p_j, w)$
    then
      $S_j(p_j, w', w)$
    end
  ...
end

An Event-B event has parameters p_i, a guard $G_i(p, v)$ (often represented as conjunction of individual guards), and a before-after predicate $R_i(p_i, v', v)$ (and $S(v', v)$ for an action of a concrete model), relating the new state v' to the previous state v. Initialisation is a special case of the event which provides the initial system state. The essential part of the Event-B method is a gradual, stepwise detailisation of the model using the refinement technique.

2.2 Refinement Patterns

The refinement pattern (initially introduced in [8]) is a set of rules describing how an output specification is produced for some input specification. Following the structure of Event-B, patterns are made of rules describing transformations on variables, invariants and events.

Definition *Refinement.* The fact that S_1 is refined by S_2 is written as $S_1 \sqsubseteq S_2$. The $\sqsubseteq$ relation is reflexive (a specification is a valid refinement of itself), transitive and antisymmetric ($S_1 \sqsubseteq S_2 \wedge S_2 \sqsubseteq S_1 \Rightarrow S_1 = S_2$).

Definition *Refinement pattern (or pattern).* Let S be the universe of specifications. Function $p : S_1 \rightarrow S_2$ where $S_1 \subseteq S$ and $S_2 \subseteq S$ such that $\forall s \cdot (s \in S_1 \Rightarrow s \sqsubseteq p(s))$ is called *refinement pattern*.

A pattern is *correct* if for any input specification it produces a correct refinement of the specification. A correct refinement is understood as a well-formed specification that is a refinement of its abstract specification.

We use the Event-B well-formedness and refinement conditions as the basis for formulating pattern correctness conditions [14]. A pattern can be proved to be correct for a whole class of input specifications. In the most general case, this class covers all Event-B specifications. Some restrictions are introduced when formulating a pattern by introducing parameters and requirements. Additional restrictions may arise when trying to prove the pattern correctness.

Class of specifications accepted by a given pattern will be referred to as pattern input specification class (ISC). We can say that a pattern relates its ISC to another class of specifications. Proving pattern correctness involves demonstrating that these two classes are linked by the refinement relation. To prove pattern correctness, the following conditions need to be shown to have been satisfied:

PAT_REQ_FIS	$\exists p \cdot RQ(p)$
PAT_FIS_INI	$\exists w' \cdot S(w, w')$
PAT_INV_INI	$S(w, w') \Rightarrow J(v, w')$
PAT_REF_FIS	$I(v) \wedge J(v, w) \wedge H(w) \Rightarrow \exists w' \cdot S(w, w')$
PAT_REF_GRD	$I(v) \wedge J(v, w) \wedge H(w) \Rightarrow G(v)$
PAT_REF_INV	$I(v) \wedge J(v, w) \wedge H(w) \wedge S(w, w') \Rightarrow$ $\exists v' \cdot (R(v, v') \wedge J(v', w'))$
PAT_REF_DLK$_i$	$G_i \Rightarrow \bigvee H_j(w)$
PAT_NEW_INV	$I(v) \wedge J(v, w) \wedge H(w) \wedge$ $S(w, w') \Rightarrow J(v, w')$
PAT_NEW_DIV	$I(v) \wedge J(v, w) \wedge H(w) \wedge S(w, w') \Rightarrow$ $V(w) \in \mathbb{N} \wedge V(w') < V(w)$

Where v and w are the abstract and concrete variables, $R(v, v')$ and $S(w, w')$ are the abstract and concrete before-after predicates, $I(v)$ and $J(v, w)$ are abstract and concrete invariants, and $G(v)$ and $H(w)$ are abstract and concrete event guards. Variant $V(w)$ is an expression decreased by all the new events. Finally, RQ is the conjunction of pattern requirements.

The main difference between proving the correctness of a pattern and that of a refinement is that rather than

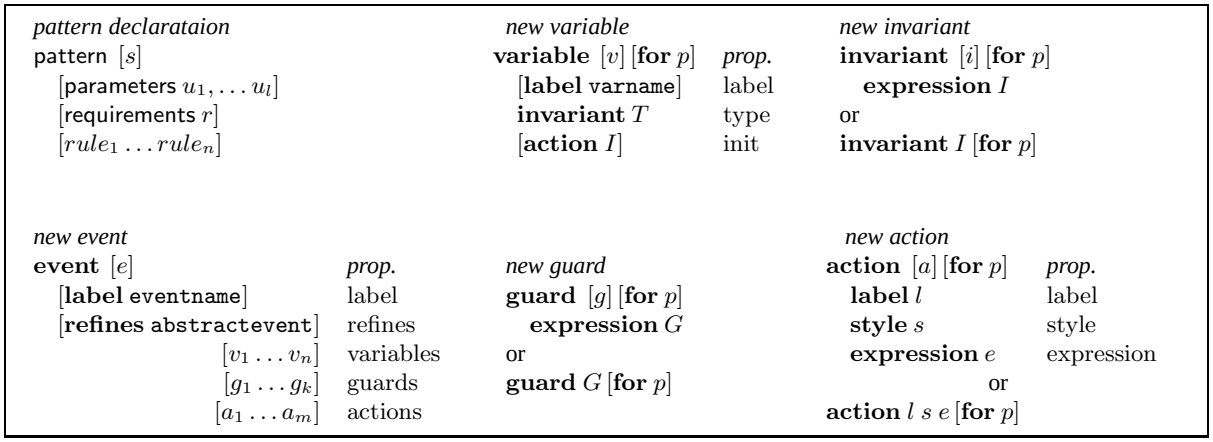

Figure 1. Summary of the notation of Event-B refinement patterns

substituting concrete expressions for invariants, guards and before-after predicates, we operate on variables representing the abstract invariant, guards and before-after predicates. To prove refinement conditions, we substitute concrete predicates for H, J, S and V, while I, G and R remain undefined.

2.3 Tool Support

Tool support is essential for the pattern mechanism proposed. While applying patterns manually is laborious and error-prone, this operation is easy to mechanise. We have implemented a plug-in to the RODIN Event-B platform [16]) which adds the pattern functionality to the platform. Once installed, it provides the environment for working with refinement patterns - selecting, editing and applying patterns in an automatic or semi-automatic manner. Owing to the open architecture of the platform, the plug-in integrates into its interface and has an intuitive user interface. It fully supports the pattern language discussed in this paper and relies on the XML notation for pattern input and editing. All the patterns presented in this paper have, in fact, been developed using the plug-in and are available for downloading, as well as the plug-in itself, from [7].

The plug-in also provides an interface for importing patterns from an online pattern library which can be updated by pattern developers. A pattern would come together with information about its purpose and origin and guidance on applying it in development.

3 Patterns for software fault tolerance

With the growing complexity of software, mistakes related to it (e.g. in applications and their components, mismatches between them, in data and supporting software, and service degradation) are becoming a major source of system downtime. The two major techniques developed for tolerating software bugs are recovery blocks and N-version programming [15, 2]. These techniques and their variations have been successfully used in a number of critical industrial applications. For example, book [17] reports successful experience of employing such schemes in the aerospace, transportation and atomic energy industries.

As increasingly complex computer-based systems proliferate in new domains of our life, we clearly need to employ these fault tolerance techniques in new settings and on a wider scale. Moreover, since society now critically depends on computer-based systems in many areas, we need to be able to apply these techniques in a rigorous and predictable way. The challenge here is to support formal stepwise development of fault tolerant systems.

To this end we propose an approach to developing software fault tolerance refinement patterns as the main means of ensuring cost-effectiveness and correctness of employing software fault tolerance mechanisms in formal system design. This approach is supported by a tool which allows us to collect a set of patterns to be applied as necessary during the development process.

Developing a refinement pattern is not a trivial task. First, it should be generic enough to express reusable refinement steps that support general concepts useful in different contexts and development approaches. Secondly, a pattern has to be proven to guarantee that the models produced are well-formed and are the correct refinements of abstract input models. In this paper we focus on two concrete software fault tolerance refinement patterns inspired by well-known design solutions. It is clear that the approach can be applied to other software fault tolerance mechanisms.

4 Constructing a Refinement Pattern

Many standardised design solutions (e.g. design patterns) can be used as a basis for constructing refinement patterns. It is not necessarily a one-to-one mapping and there are design solutions that are so abstract they cannot be expressed as refinement patterns. One example is the proxy pattern ([6]), which is an abstract rule for structuring a complex system. The pattern is so abstract that it cannot be described in a general form using the refinement patterns mechanism. However, there can be any number of specialised versions of the pattern formulated for particular kind of systems which are expressable as refinement patterns.

A legacy or current formal development is an important source of ideas for refinement automation. In many cases, concrete refinement steps are but implementation of a more general concept. Application of the concept to a whole range of possible abstract systems results in a refinement pattern. A side effect - a deeper understanding of the concept used in development through it formalisation - is important by itself.

A pattern designer is likely to have a choice of a level of pattern presentation. A refinement pattern can be more abstract and thus be applicable to a wider range of input specifications or more specific and restricted to a narrower domain. More abstract patterns are generally easier to apply as they put less restrictions on the form of an abstract specification. One of the objectives during pattern development is to find a balance between pattern generality and details in describing its functionality.

The first step of a pattern design is identification of the possible pattern parameters, their types and restriction. This stage defines the set of possible abstract specifications that can be transformed by this pattern which loosely corresponds to the notion of the problem domain of a design pattern (solution). For complex patterns it is a non-trivial task to identify the weakest set of patterns requirements. The strategy is to start with a minimal set of requirements and use proof obligations to add additional restrictions that are also necessary to demonstrate pattern correctness.

It is important to carefully identify major building blocks of a pattern: new variables and events declared by the pattern, top-level matching blocks, refinements of abstract variables and events. These serve as a skeleton around which further details are added. Proofs will help to identify missing or wrong rules.

Once there is sufficiently detailed description of a refinement pattern, proof obligations for pattern correctness can be generated. With a theorem prover, many proof obligations are discharged automatically. A proof obligation not discharged automatically may indicate a mistake in pattern thus playing an important role in pattern design. A structure of a proof obligation often provides valuable information on how to rectify a problem in the patterns. Typically, several iterations are needed to produce a correct pattern.

5 Recovery Block Pattern

This pattern helps to develop software capable of tolerating software faults by introducing N alternates designed diversely following the ideas from [15]. Checkpointing is used to save the state before executing an alternate so that the results of unsuccessful execution can be discarded. An alternate execution is followed by checking an acceptance test. If the test is passed then the result of the current alternate is used as the final result. Otherwise, the state is rolled back and another alternate is activated. If no alternate is available, an exception is propagated.

The pattern takes as input a model with two events. One of the events is a specification of the desired behaviour. The other event is the connection to some external recovery or abortion mechanism. During instantiation, the pattern also asks for the number N of behaviour block instances (alternates).

Further refinements should diversify designs of behaviour alternates (e.g. by enforcing the use of different solutions and by involving different developers) and adapt or make more concrete the test conditions. A good starting point for applying this pattern is a specification with non-deterministic before-after predicates. The conjunction of all before-after predicates of an abstract behaviour event is used by the pattern as the acceptance test. The pattern has three parameters

pattern $recblock$	req_$typing$ $b \in \mathbf{Event} \wedge n \in \mathbb{N}$
parameters b, n	req_$notempty$ $card(b.\text{actions}) > 0$
	req_$notzero$ $n > 0$

Here b is an abstract event specifying the desired system behaviour, a is an abortion event and n is a number of recovery blocks. The pattern requirements state the typing of the parameters and also state that the behaviour event contains at least one action and that the number of alternates is not zero:

This pattern can be applied to any specification with at least two events, one of which must be not empty. The pattern makes no additional assumptions about event bodies, guards and parameters as the pattern is general enough to handle all the possible cases.

The pattern introduces two new variables (these variables appear in the output specification) to model control flow for the new events. Variable br defines the currently active behaviour block (alternate). When its value goes beyond the allowed block index, it indicates failure of all the blocks. Variable st indicates the current stage: checkpoint ($st = 0$), block ($st = 1$) or acceptance test ($st = 2$).

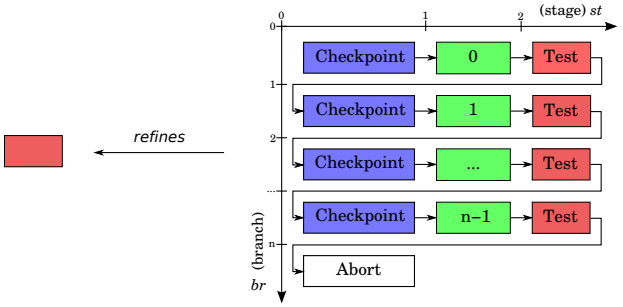

Figure 2. The Recovery Block pattern. The checkpoint and alternates are modelled as new events, the *Test* block refines the behaviour abstract event.

variable br
  invariant $br \in 0..(n+1)$
  action $br := 0$

variable st
  invariant $st \in 0..2$
  action $st := 0$

The pattern models checkpointing by extending system state with new variables for hold intermediate results produced by the alternates. If the result of an alternate fails the acceptance test, the state extension is disregarded. When test succeeds, the state is used as the final result. This approach allows us to introduce checkpoints without knowing the whole system state. The following pattern fragment creates a copy of each variable assigned in the event b:

forall a where $a \in b$.actions
  variable $cpvar$ **for** $recblock$
    label cp_a.variable.name
    invariant $cpvar \in a$.variable.type
    action $cpvar$
      a.variable.init.style a.variable.init.expr

The pattern fragment below creates a checkpoint event which saves the current values of the variables updated in the event b. This event is enabled when $st = 0$:

event $chkpt$
  guard $st = 0$
  forall a where $a \in b$.actions
    action cp_a.variable.name $:=$ a.variable.name
  action $st := 1$

The event advances the stage variable st so that a currently selected alternate is enabled. An alternate contains the same set of actions as the abstract event b. These actions assign to the copies of the abstract variables updated in b. Although, an alternate is formally not a refinement of b, it is related through the actions. A designer has the choice of changing alternates behaviour just after applying the pattern or keeping them intact and using refinement to gradually introduce specialisation. In the latter case, the actions derived from the actions of b serve as an abstract specification for further refinements. To allow for meaningful refinements these actions must be non-deterministic. The next pattern fragment produces n events representing recovery block alternates:

forall i where $i \in 1..n$
  event alt
    label b_alt_i
    guard $st = 1$
    guard $br = i - 1$
    guard b.guards
    variable b.variables
    forall a where $a \in b$.actions
      action cp_a.variable.name a.style a.expr
    action $st := 2$

Note the guard $br = i - 1$ selecting the current alternate and action $st := 2$ enabling the acceptance test. The acceptance test event checks if the alternate has succeeded and, if it is so, uses the its result as the final result. The acceptance test must refine b since it is the only event which is allowed to update inherited abstract variables which the abstract version of b used to produce the result. In other words, an input specification is transformed in such a way that parts which the pattern is not aware about are not effected.

The acceptance test is computed automatically by the pattern from the abstract event b. In English, the acceptance can be informally formulated as *any result that agrees with the specification of the abstract event b is acceptable*. To give the exact meaning to *agrees with* we use the before-after predicates of the abstract event b:

guard $st = 2$ **for** b
forall a where $a \in b$.actions
  guard [cp_a.variable.name a.style a.expr] **for** b
  action a.variable.name $:=$ cp_a.variable.name **for** b

We also have to address the case when the acceptance test fails. For this we declare a new event and use a guard which is the opposite of the acceptance test rule. One of the responsibilities of this event is to advance the br variable so that a new alternate is used next time.

event $test$
  label b_test_fail
  guard $st = 2$
  guard $\bigvee_{a \in b.\text{actions}} \neg$ [cp_a.variable.name a.style a.expr]
  action $br := br + 1$
  action $st := 0$

$\bigvee_{a \in b.\text{actions}} \ldots$ and other generalised versions of operators are syntax shortcuts. The actual expression for this guard is made of three nested statements and expressed in 8 lines of XML notation.

Since we have only n alternates with indices $0..n-1$, a state where $br = n$ indicates that all the alternates have failed to produce an acceptable result. To cover the case of $br = n$ the patterns produces a new events which simply uses the abstract event b behaviour to produce some "safe" result.

event $fail$
  label b_fail
  refines b
  guard $br = n$ **for** f
  guard b.guards
  variable b.variables
  action b.actions

5.1 Recovery Block Pattern Correctness

In this section we demonstrate that the Recovery Block pattern indeed produces valid refinements for any input specification to which it can be applied. Here we write out and analyse proof obligations manually. Most of this can be handled by a tool and we are working on adding support for generating proof obligations and automatically discharching them with the platform theorem prover.

Pattern requirements must allow for a non-empty set of parameters

$$\exists(b, f, n) \cdot (\mathsf{req\_}typing \ \wedge \mathsf{req\_}notempty \ \wedge \mathsf{req\_}notzero \ )$$

This proof obligation (PO) can be discharged by asserting that set **Event** is nonempty and there exist an event with at least one action.

Declarations of br and st result in the following proof obligations

$$\begin{array}{l}\exists br' \cdot (br \in 0..(n+1) \wedge br' = 0)\\ br \in 0..(n+1) \wedge br' = 0) \Rightarrow br' \in 0..(n+1)\\ \exists st' \cdot (st \in 0..2 \wedge st' = 0)\\ st \in 0..2 \wedge st' = 0 \Rightarrow st' \in 0..2\end{array}$$

which are trivially true.

The pattern introduces new system variables supporting checkpointing. For each variable updated in the event b a new variable is created with the same type and initial state. To express this, the pattern uses the forall a where $a \in b$.actions construct. Consequently, related proof obligation are in the form $\forall a \ldots$:

$$\begin{array}{l}\forall a \cdot (a \in b.\text{actions} \Rightarrow \exists c' \cdot (c \in Tp(a) \wedge [c \ St(a) \ In(a)]))\\ \forall a \cdot (a \in b.\text{actions} \Rightarrow [c \ St(a) \ In(a)] \Rightarrow c' \in Tp(a))\end{array}$$

where $[vse]$ is a before-after predicate of a B action made from variable v, action style s ($:=$, $:\in$ and $|\in$) and expression s. Also, the following shortcuts are used: $c = cpvar$, $St(a) = a$.variable.init.style, $Tp(a) = a$.variable.type and $In(a) = a$.variable.init.expr. The proof obligations above can be simplified by removing quantifier $\forall a$ and treating a as a free variable. To do the proof we use information about the abstract variables from which the copied variables are derived

$$\begin{array}{l}a \in b.\text{actions} \Rightarrow \exists v' \cdot (v \in Tp(a) \wedge [v \ St(a) \ In(a)]) \vdash\\ \quad a \in b.\text{actions} \Rightarrow \exists c' \cdot (c \in Tp(a) \wedge [c \ St(a) \ In(a)])\\ a \in b.\text{actions} \Rightarrow [v \ St(a) \ In(a)] \Rightarrow v' \in Tp(a)) \vdash\\ \quad a \in b.\text{actions} \Rightarrow [c \ St(a) \ In(a)] \Rightarrow c' \in Tp(a))\end{array}$$

The proof obligations above are trivially correct as left and right parts differ only in the names of free variables. Note, that the proof covers the general case of creating variable copies.

The checkpoint event initialises sub-states used by the recovery blocks. The action updating st gives rise to the following trivial POs:

$$\begin{array}{l}I(v) \wedge st \in 0..2 \wedge st = 0 \Rightarrow \exists st' \cdot (st \in 0..2 \wedge st' = 1)\\ I(v) \wedge st \in 0..2 \wedge st = 0 \wedge st' = 1 \Rightarrow st' \in 0..2\end{array}$$

Initialisation of checkpoint variables uses the forall statement and hence the universal quantifier appears in the proof obligations:

$$\begin{array}{l}\forall a \cdot (a \in b.\text{actions} \Rightarrow I(v) \wedge c(a) \in Tp(a) \wedge v(a) \in Tp(a) \wedge\\ \quad st = 0 \Rightarrow \exists c(a)' \cdot (c(a) \in Tp(a) \wedge c'(a) = var(a)))\\ \forall a \cdot (a \in b.\text{actions} \Rightarrow I(v) \wedge c(a) \in Tp(a) \wedge v(a) \in Tp(a) \wedge\\ \quad st = 0 \wedge c'(a) = var(a) \Rightarrow c'(a) \in Tp(a))\end{array}$$

where $c(a) = \mathsf{cp\_}a$.variable.name, $v(a) = a$.variable.name and $St(a)$, $In(a)$ and $Tp(a)$ as defined above. The quantifier can be dropped and with the properties of the original variables as hypothesis it is trivial to discharge these POs.

The pattern fragment creating the recovery blocks employs forall statements. The outer one runs through all the recovery block indices. while the inner one creates a new action for each action in the abstract event b. The proof obligations are the following:

$$\begin{array}{l}\forall i \cdot (i \in 1..n \Rightarrow \forall a \cdot (a \in b.\text{actions} \Rightarrow\\ \quad I(v) \wedge c \in Tp(a) \wedge st = 1 \wedge br = i-1 \Rightarrow\\ \quad\quad \exists c' \cdot (c \in Tp(a) \wedge [c \ St(a) \ Ex(a)])))\\ \forall i \cdot (i \in 1..n \Rightarrow \forall a \cdot (a \in b.\text{actions} \Rightarrow\\ \quad I(v) \wedge c \in Tp(a) \wedge st = 1 \wedge br = i-1 \wedge\\ \quad\quad [c \ St(a) \ Ex(a)] \Rightarrow c' \in Tp(a)))\end{array}$$

The proofs are easy to do for the general case of some index i and some action a and the quantifiers can be removed. The we know that the abstract actions are well-formed we use this information as hypothesis to discharge the proof obligations. The case for the action assigning st is trivial.

In the acceptance test fragment defines actions replacing the abstract actions of event b. We have to prove that under the given conditions each such action refines its abstract counterpart

$$\forall a \cdot (a \in b.\text{actions} \Rightarrow Tp(a) \Rightarrow \exists v(a)' \cdot (c(a) \in Tp(a) \wedge v(a)' = c(a)))$$
$$\forall a \cdot (a \in b.\text{actions} \Rightarrow I(v) \wedge [c(a)\ St(a)\ Ex(a)] \wedge v(a) = c(a) \Rightarrow \exists v' \cdot ([v(a)\ St(a)\ Ex(a)] \wedge va(a) \in Tp(a)))$$

These POs are trivially correct. For concrete version of event b we have to demonstrate that the new guard is stronger than its abstract counterpart.

$$I(v) \wedge J(v,w) \wedge H(w) \Rightarrow G(v)$$

It is indeed so, as the pattern fragment strengthens the guard with additional conditions.

For the new event $test$ we have several obvious trivial POs due to the action $br := br + 1$ and $st := 0$.

To prove the non-divergence of the new events introduced by the pattern we have to demonstrate there exists such $V \in \mathbb{N}$ that it is decreased by all the new events.

$$I(v) \wedge J(v,w) \wedge H(w) \wedge S(w,w') \Rightarrow V(w) \in \mathbb{N} \wedge V(w') < V(w)$$

Let $V = (n+1) * 3 + 2 - (br * 3 + st)$ and $T = st \in 0..2 \wedge br \in 0..(n+1) \wedge n \in \mathbb{N} \wedge n > 0$

Condition $T \Rightarrow V(w) \in \mathbb{N}$ holds since $\max(br * 3 + st) = (n+1)*3+2$. To prove that all the events decrease V we have to demonstrate that the following conditions hold

$$T \wedge st = 0 \wedge st' = 1 \wedge br' = br \Rightarrow V(st', br') < V(st, br)$$
$$\forall i \cdot (i \in 1..n \Rightarrow T \wedge st = 1 \wedge st' = 2 \wedge br' = br \Rightarrow V(st', br') < V(st, br))$$
$$T \wedge st = 2 \wedge st' = 0 \wedge br' = br + 1 \Rightarrow V(st', br') < V(st, br)$$

The first two increment st leaving br unchanged and hence decrease the variant expression since V monotonously decreasing function. The $test$ event resets st to zero but this is compensated by the increment of br.

To prove relative deadlock freeness we have to demonstrate that for any abstract state in which guard G is enabled there is a a state in the refined version in which at least on of the concrete guards is enabled. The disjunction of the concrete guards is

$$\bigvee \mathsf{H_j(w)} \quad st = 0 \vee (st = 1 \wedge G_b \wedge \bigvee_{i \in 1..n} br = i - 1) \vee (st = 2 \wedge \bigwedge_{a \in b.\text{actions}} act) \vee (st = 2 \wedge \bigvee_{a \in b.\text{actions}} \neg act) \vee br = n \wedge G_b$$

where G_b is the guard of event b and $act = [\mathsf{cp}\_a.\text{variable.name}\ a.\text{style}\ a.\text{expr}]$. The above simplified to

$$\bigvee \mathsf{H_j(w)} \quad st = 0 \vee \left(st = 1 \wedge ((G_b \wedge \bigvee_{i \in 0..(n-1)} br = i) \vee (br = n \wedge G_b))\right) \vee st = 2$$

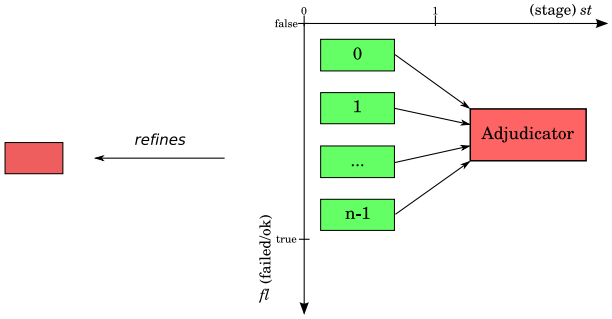

Figure 3. The NVP pattern. Versions are new events and the *adjudicator* refines the abstract behaviour event.

and we have to prove that

$$\mathsf{PAT\_REF\_DLK_b} \quad G_b \Rightarrow G_b \wedge (\bigvee_{i \in 0..(n-1)} br = i \vee br = n)$$

which is true since $(\bigvee_{i \in 0..(n-1)} br = i \vee br = n)$ always holds.

6 N-Version Programming Pattern

N-Version Programming is a software engineering method for tolerating mistakes in software implementation by using a number of functionally-equivalent versions developed independently according to common requirements or specifications [2]. The method is based on selecting the majority result from the outputs of all the versions.

Our NVP pattern takes two arguments - an event b and number of blocks n. The result of the pattern application is a set of n behaviour blocks and the adjudicator which refines b.

pattern nvp
parameters b, n
req_*typing* $b \in \mathbf{Event} \wedge n \in \mathbb{N}$
req_*grtone* $n > 1$

Variable st defines the major state evolution stage of a system produced by the pattern: 0 is for collecting results from individual blocks, 1 for voting and 2 when the final result is available. The pattern introduces a Boolean variable fl indicating inability to find a dominating (a result with 50% + 1 votes).

variable st
invariant $st \in 0..2$
action $st := 0$

variable fl
invariant $fl \in \mathbb{B}$
action $fl := FALSE$

All the N versions produce their results independently and thus they must operate on disjoint state spaces. A simple solution is to introduce a function from a block id into a

state associated with the block and let each block modify its own state using the function. Such approach, however, results in several unattractive properties of the pattern. First, it introduces a variable shared by all the blocks - the state function variable - and there is nothing preventing an inexperienced designer from accidentally mixing block states, both in a model and in implementation. It also prohibits an automated refinement into an efficient, concurrent implementation. Second, and more important, a refinement produced by such pattern may not easily legible. State of a block is likely to be a complex type. Dealing with such involved structures is much more difficult than with individual variables. Hence decided to have a separate set of variables for each behaviour block. This complicates the pattern definition and makes the pattern correctness proofs slightly more difficult but the result is refinement pattern which easy to use. After applying it, a designer gets N new events which are similar to the abstract event to which the pattern was applied.

Each behaviour block is attached a Boolean variable indicating that the block has finished and the voter event can use the current result. This variable can also be used to disable permanently faulty blocks, although we do not do this in the current version to keep the pattern general.

The body of a behaviour block is almost an exact copy of event b with the only difference being actions assigning values to copies of the original abstract variables. Each behaviour block has its own set of copied variables.

forall i where $i \in 1..n$
  variable rd
    label `rd_`i
    invariant $rd \in \mathbb{B}$
    action $rd := FALSE$
  event alt
    label `alt_`i
    guard $st = 0$
    guard $rd = FALSE$
    forall a where $a \in b.\text{actions}$
      variable cp
        label $a.\text{variable.name\_}i$
        invariant $cp \in a.\text{variable.type}$
        action $cp\ a.\text{variable.init.style}\ a.\text{variable.init.expr}$
      action $cp\ a.\text{style}a.\text{expr}$
    action $rd := TRUE$

When the results from all the blocks are available, the voter can select the final result. To produce a scalable solution we have to aggregate individual variables used in different blocks into a single variable, which is a function from a block id into the block state. We do not expect designers to change this part of the specification thus we are free to use the most suitable approach here.

variable rs **for** nvp
  label b`_result`
  invariant $rs : \mathbb{N} \hookrightarrow (\bigtimes_{a \in b.\text{actions}} a.\text{variable.type})$
  action $rs := \varnothing$

where $(\bigtimes_{a \in b.\text{actions}} a.\text{variable.type})$ is the type of a tuple $(v_1, v_2, \ldots v_n)$ used to store all the variable assigned in event b. The general cartesian product is a syntax sugar; interested readers will find the actual representation in the pattern source available from the plug-in web page [7].

The following event constructs the function of results from the result of the individual blocks

event $accum$
  label b`_collect`
  guard $st = 0$
  guard $\bigwedge_{i \in 1..n}$ `rd_`$i = TRUE$
  action $rs := \bigcup_{i \in 1..n} \{i \mapsto (\mapsto_{a \in b.\text{actions}}\ var\_i)\}$
  action $st := 1$

where $var\_i = a.\text{variable.name\_}i$.

The adjudicator event refines abstract event b. The pattern adds additional guards, parameters and actions and also changes the abstract action. The parameters are used as local variables which help to select the final result. The event guards describes a simple voting protocol and there is an action indicating if the winning result has got the majority of votes.

Parameter k is the index of the winning result, parameters $a.\text{variable.name\_t}$ are used to extract solution from function rs.

variable k **for** b
  invariant $k \in dom(rs)$
forall a where $a \in b.\text{actions}$
  variable t **for** b
    $a.\text{variable.name\_t}$
    invariant $t \in a.\text{variable.type}$

The first guard makes the event enabled at stage 1, the next one selects k such that k is an index of a winning solution (k is the index of a winning solution if for all j different from k the number of indices pointing at the same solution as k is greater or equal to the number of solutions pointed to by j) and the last guard binds parameters to the values of the solution.

guard $st = 1$ **for** b
guard $\forall j \cdot (j \in dom(rs) \wedge j \neq k \Rightarrow$
  $card(rs^{-1}[\{rs(k)\}]) \geq card(rs^{-1}[\{rs(j)\}]))$ **for** b
guard $(\mapsto_{a \in b.\text{actions}}\ a.\text{variable.name\_t}) = rs(k)$ **for** b

In the event body abstract action are replaced with action copying values from the parameters used to extract the solution. The stage variable is advanced to indicate that the final result is available and for all the blocks the status variable is to *false* to prepare for a possible next iteration.

forall a where $a \in b.\text{actions}$
  action $a.\text{variable.name} := a.\text{variable.name\_t}$ **for** b
action $st := 2$ **for** b
forall i where $i \in 1..n$
  action $\texttt{rd\_}i := FALSE$ **for** b
action $fl := bool(card(rs^{-1}[\{rs(k)\}]) < (n/2+1))$ **for** b

Here fl is a Boolean flag indicating whether the solution has got the majority of votes or not.

6.1 NVP Pattern Correctness

Most proof obligations for this pattern are trivially discharged and the techniques employed are the same as those used for the Recovery Block pattern.

The only non-trivial part is to demonstrate that the voting event refines the abstract behaviour event. In other words, a solution selected by the voting event must satisfy the specification of the abstract event b. However, since the pattern does not itself produce diverse version blocks and further refinements of version blocks satisfy the abstract event b specification by definition of refinement, the voting mechanism has no effect on the selection of the result. It is enough to demonstrate that the values carried through the rs function are the results collected from version blocks. It is obviously so, since the function is only assigned in the event *accum*, which copies version results.

7 Applying Patterns

Our experience indicates that the refinement pattern mechanism can make a considerable impact on the development process as patterns support reuse and make development easier and less error-prone.

One of the most attractive features of refinement patterns is that, if supported by the right tool, pattern application is almost instantaneous and straightforward. Various refinement paths can be investigated, rather than by investing a considerable time and modelling effort, just by selecting different patterns. If a result is unsatisfactory, the pattern is undone to allow trying a different one. This is a considerable advantage over manual refinement, where a developer would be reluctant to redo modelling steps once committed to a particular solution.

Reading and applying refinement patterns is much easier than writing them, allowing us to draw a line between a formal method expert who designs high-quality reusable patterns and an engineer using patterns to design a system model. The fact that applying patterns does not require a high level of expertise in formal methods can contribute to a wider adoption of formal modelling as a cost-effective software engineering technique for safety-critical and dependable systems.

The effect of patterns on the development process depends on the number and quality of available patterns. We consider pattern correctness as a sufficient measure of pattern quality: while there are some correct patterns which do not produce any useful transformations, this measure will not allow any patterns which construct invalid refinements. Unlike concrete refinement steps, patterns are designed to be reusable. An important part of the pattern mechanism is its capacity to look for patterns which can do the required transformation. The current tool implementation offers some support for importing from an online pattern library using a dialogue, with patterns sorted into a tree according to their functionality. We are now working on a finer pattern search mechanism that would allow the engineer to search for a pattern by specifying a design goal.

A refinement pattern is a complete, self-sufficient unit of modelling that can be communicated between developers to support reuse and experience sharing. It can do to formal model development what components and libraries do to program development. With an extensive pattern library, the entire system design can be performed through employing third-party patterns with some custom logic filled in places.

The proposed software fault tolerance patterns are designed to be applied as part of formal system development. This should be followed by further refinement steps which will specify recovery block alternates and NVP versions. This approach fits in well with stepwise incremental system development, as different parts of models are to be refined separately and independently. We realise that it is very unlikely that this will achieve the complete independence of the version/alternate failure modes [13], but the subsequent development steps can be enriched by enforcing the use of different solutions and formalisms, and by involving different developers to diversify their designs.

Using nested units of design and execution to build systems is the main way of dealing with system complexity, and, in particular, of ensuring system fault tolerance by defining error containment and error recovery units ([15, 2]). Composing patterns (i.e. their sequential application) is a very useful mechanism for creating nested units of structuring and development in the course of stepwise system development. It is fairly easy to compose the two proposed software fault tolerance patterns, with several compositions possible, including iterative application of the Recovery Block pattern to obtain nested recovery blocks and using the NVP pattern to refine one of the recovery block alternates.

8 Conclusion

The proposed mechanisms will help to build systems that would tolerate faults of several types. First of all, these would include mistakes made in the later refinement phases

by developers working on different versions/alternates as well as those in coding different versions/alternates. Due to diversity in refinement and coding, faults in the run time environment (e.g. OS, middleware) can be tolerated as well. Moreover, when versions/alternates are distributed (e.g. as in [10]) the proposed patterns can help to tolerate faults of a wider class, including hardware crashes.

The two patterns presented in the paper, along with a number of other refinement patterns, have been applied in the development of the Ambient Campus case study within the ICT RODIN Project [9]. In this case study we have developed several application scenarios for PDAs and smart-dust devices in which fault tolerance is essential just to achieve a reasonable usability level. In particular, we use the Recovery Block pattern to alternate between different positioning services: GPS (fails indoors), motes (fails when there are not enough motes in proximity) and, finally, WiFi-based positioning.

9 Acknowledgements

This work is supported by the FP6 ICT RODIN and FP7 ICT DEPLOY Projects, and by the EPSRC/UK TrAmS Platform Grant. Alexei Iliasov is partially supported by the ORS award (UK).

References

[1] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[2] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transaction on Software Engineering SE-11, 12 (December), 1491-1501.*, 1985.

[3] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 242–251, New York, NY, USA, 2006. ACM Press.

[4] F. Cristian. Exception Handling and Fault Tolerance of Software Faults. In M. Lyu, editor, *Software Fault Tolerance*, pages 81–107. Wiley, NY, 1995.

[5] F. C. Filho, P. H. da S. Brito, and C. M. F. Rubira. Specification of exception flow in software architectures. journal of systems and software. *Journal of Systems and Software*, (79(10)):1397–1418, 2006.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley. ISBN 0-201-63361-2, 1995.

[7] A. Iliasov. Finer Plugin Web Page. *http://finer.iliasov.org*, Last accessed: 21 Feb 2008.

[8] A. Iliasov. Refinement patterns for rapid development of dependable systems. *Proc. Engineering Fault Tolerant Systems Workshop (at ESEC/FSE, Croatia), ACM*, September 4, 2007.

[9] A. Iliasov, A. Romanovsky, B. Arief, L. Laibinis, and E. Troubitsyna. On rigorous design and implementation of fault tolerant ambient systems. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 141–145, Washington, DC, USA, 2007. IEEE Computer Society.

[10] K. H. Kim and H. O. Welch. Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Transactions on Computers*, 38(5):626–636, 1989.

[11] L. Laibinis and E. Troubitsyna. Refinement of Fault Tolerant Control Systems in B. In M.Heisel, P.Liggesmeyer, and S.Wittmann, editors, *Computer Safety, Reliability, and Security - Proceedings of SAFECOMP 2004*, number 3219 in Lecture Notes in Computer Science, pages 254–268. Springer-Verlag, Sep 2004.

[12] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 418–427, New York, NY, USA, 2000. ACM Press.

[13] B. Littlewood, P. Popov, and L. Strigini. Modeling software design diversity: a review. *ACM Comput. Surv.*, 33(2):177–208, 2001.

[14] C. Metayer, J.-R. Abrial, and L. Voisin. *Rodin Deliverable D7: Event B language*. Project IST-511599, School of Computing Science, University of Newcastle, 2005.

[15] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering. IEEE Press*, SE-1(2):220–232, 1975.

[16] Rodin. Rigorous Open Development Environment for Complex Systems. *IST FP6 STREP project, http://rodin.cs.ncl.ac.uk/*, Last accessed: 18 June 2007.

[17] U. Voges, editor. *Software diversity in computerized control systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1988.