

Structuring Specifications with Modes

Alexei Iliasov, Alexander Romanovsky

Center for Software Reliability

Newcastle University

England, United Kingdom

Email: {alexei.iliasov, alexander.romanovsky}@newcastle.ac.uk

Fernando Luís Dotti

Faculty of Informatics

Pontifical Catholic University of Rio Grande do Sul

Porto Alegre - RS - Brazil

Email: fernando.dotti@pucrs.br

Abstract—The two dependability means considered in this paper are rigorous design and fault tolerance. It can be complex to rigorously design some classes of systems, including fault tolerant ones, therefore appropriate abstractions are needed to better support system modelling and analysis. The abstraction proposed in this paper for this purpose is the notion of operation mode. Modes are formalised and their relation to a state-based formalism in a refinement approach is established. The use of modes for fault tolerant systems is then discussed and a case study presented. Using modes in state-based modelling allows us to improve system structuring, the elicitation of system assumptions and expected functionality, as well as requirement traceability.

Keywords—operation modes; fault-tolerance; formal specification; Event-B.

I. INTRODUCTION

Systems are dependable if they deliver service that can be justifiably trusted [1]. Building such systems is a challenging task, typically conducted by employing various dependability means. In this paper we are particularly interested in the means of two types: rigorous design and fault tolerance.

Rigorous design (or fault prevention) is often used to justify system trustworthiness by preventing introduction of faults into system. This can be done by employing formal modelling and analysis. The known problem with this approach is its scalability. A way to improve it is through the development of abstractions and formal techniques tailored to classes of systems.

System dependability cannot be achieved by only trying to build perfect systems, any critical system has to face abnormal situations (including malfunctioning devices, wearing hardware and software defects) and deal with them properly. This is achieved by integrating appropriate fault tolerance means into the system. Unfortunately the situation is not satisfactory here: as reported by F. Cristian [2], field experience with telephone switching systems showed that up to two thirds of system failures were due to design faults in exception handling or recovery algorithms. Other evidences of inadequate use or construction of fault-tolerance mechanisms are reported in [3].

This work is partially supported by the ICT DEPLOY IP and the EPSRC/UK TrAmS platform grant. Fernando L. Dotti is supported by CNPq/Brazil grant 200806/2008-4.

Several authors have investigated fault-tolerance modelling using different specification formalisms and verification approaches (e.g. [4], [5]). However, the identification and support of suitable abstractions for formal design of fault tolerant systems is still an open issue. Such abstractions have to, at the one side, be amenable to representation using a formal specification language, and, on the other side, offer the way to model and reason about (i) states: the characterization of normal and erroneous states is inherent to fault tolerant systems; (ii) structure: separation of normal and abnormal (fault tolerant) behaviour is to be supported, as well as the representation of control structures for different tolerance mechanisms; and (iii) system properties: the statement of system properties under different working conditions (addressing fault assumptions) should be supported.

In this paper the concept of ‘operation mode’ [6] is revisited. We use modes to structure system specification to facilitate rigorous design and to integrate fault tolerance. Since modes appear in different types of systems, such as real-time [6], avionic and space [7], [8], the approach is useful for building wide classes of dependable systems.

We use term mode in the same sense as [6]: both as partitions of the state space, representing different working conditions of the system, and as a way to define control information, structuring system operation. In Section II, modes are defined to allow the modeller to state the property that must be respected, called guarantee, in each working system condition, called assumption. In Section III, mode refinement is discussed, allowing detailisation of the mode system. The use of modes together with a state-based formal method is discussed in Section V. Mode refinement is performed hand in hand with the refinement of the respective formal model and allow layered definition and reasoning about properties. This helps to trace properties to requirements. Refinement also offers a strategy to obtain a correct implementation from the formal model. Theorem proving strategies and tools sometimes offer an attractive option to model-checking as they avoid the state-space problem. Section IV discusses the use of modes in the design of fault-tolerant systems. Section VI exemplifies the ideas with the model of a cruise control system. Related work and conclusions are presented in Sections VII and VIII.

II. OPERATION MODES

Operation modes help to reason about system behaviour by focusing on the system properties observed under different situations. In this approach, a system is seen as a set of modes partitioning the system functionality over differing operating conditions. The term *assumption* is used to denote the different operating conditions and *guarantee* denotes the functionality ensured by the system under the corresponding assumption. A system may switch from one mode to another in a number of ways characterised by *mode transition*.

A mode is thus a pair A/G where $A(v)$ is an assumption, a predicate over the current system state, $G(v, v')$ is the guarantee, a relation over the current and next states of the system. Vector v is the set of variables, characterising a system state and constrained by an invariant $I(v)$. The purpose of an invariant $I(v)$ is to limit the possible states by excluding undesirable or unsafe states. It also defines types for variables v . To limit the scope of discussion, it is assumed that a system is only in one mode at a time. Mode overlapping and mode interference bring a number of interesting challenges that cannot be sufficiently addressed in this paper due to space limitations. Formally, it is required that mode assumptions are mutually exclusive and exhaustive in respect to a model invariant, as below. \oplus is a set partitioning operator.

$$I(v) = A_1(v) \oplus \dots \oplus A_n(v) \quad (1)$$

Mode switching is realised with mode transitions. A mode transition is an atomic step switching system from one source to one destination mode. It is convenient to characterise a mode transition by a pair of assumptions - the assumptions of source and of destination modes. Assuming that mode is assigned an index, a mode transition from A_i/G_i to A_j/G_j is a relation on mode indices $i \rightsquigarrow j$. A system starts executing one of initiating transitions $\top \rightsquigarrow k$. The transition switches the system on and places it into some system mode A_k/G_k . A system terminates by executing one of terminating transitions $t \rightsquigarrow \perp$ ¹. Mode transitions $i \rightsquigarrow \top$ and $\perp \rightsquigarrow j$ are not allowed. Also, it is required that during its lifetime a system enters at least in one operation mode and thus transition $\top \rightsquigarrow \perp$ is not possible. There can be any number of initiating and terminating mode transitions.

There are restrictions on the way mode assumptions and guarantees are formulated. The states described by a guarantee must be wholly included into valid model states:

$$I(v) \wedge A(v) \wedge G(v, v') \Rightarrow I(v') \quad (2)$$

The assumption and guarantee of a mode must be non-contradictory. I.e. a mode should permit a concrete implementation:

$$\exists v, v'. (I(v) \wedge A(v) \Rightarrow G(v, v')) \quad (3)$$

¹Not every system has to have this transition: a control system would be typically designed as never aborting.

A system is characterised by a collection of modes and a vector of mode transitions:

$$\begin{array}{l} A_1/G_1, \dots, A_n/G_n \\ i \rightsquigarrow j, \dots, k \rightsquigarrow l \end{array} \quad (4)$$

The state of a system described using operation modes is a tuple (m, v) where m is the index of a current operation mode and v is the current system state. Mode index helps to clarify how mode switching is done although it may be computed from v alone due to condition 1. The evolution of a system like above is understood as follows. While it is in some mode m the state of model variables evolves so that the next state is any state v' satisfying both the corresponding guarantee $G(v, v')$ and the modes assumption $A(v')$:

$$\boxed{\text{internal}} \frac{A_m(v) \wedge G_m(v, v') \wedge A_m(v')}{\langle m, v \rangle \rightarrow \langle m, v' \rangle}$$

If there is a mode transition originating from a current mode, the transition could be enabled to switch the system to a new mode.

$$\boxed{\text{switching}} \frac{m \rightsquigarrow n \wedge A_m(v) \wedge A_n(v')}{\langle m, v \rangle \rightarrow \langle n, v' \rangle}$$

These two activities compete with each other: at each step a non-deterministic choice is made between the two. An initiating transition is a special case: it must find an initial system state without being able to refer to any previous state:

$$\boxed{\text{start}} \frac{\top \rightsquigarrow k \wedge A_k(v)}{\langle \top, undef \rangle \rightarrow \langle k, v \rangle}$$

where *undef* denotes a system state prior to the execution of an initiating transition. System termination is addressed by the *switching* rule above. Note that all of the three rules also assume that an invariant holds in current and new states: $I(v) \wedge I(v')$. This is a corollary of conditions 1 and 2.

III. MODE REFINEMENT

Refinement is formal technique for transitioning from an abstract model to a concrete one [9]. Terms abstract and concrete are relative here: a concrete model of one step is another's step abstract model. There are a number of benefits in apply refinement in model construction: it combats complexity by splitting design process into a number of simple steps; it helps to organise the process of modelling by allowing a modeller to focus on one aspect of a model a time; it makes proofs easier as for each refinement one only has to proof the correctness of new behaviour². Refinement is a partial order relation on model universe. This relation is denoted as \sqsubseteq and it is reflexive, transitive and antisymmetric. For the operation modes mechanism the refinement technique is used to gradually evolve a system description by adding or replacing modes and transitions. Such evolution is formal in a sense that a refined model may be used in place of its abstraction. A number of refinement techniques can be used.

²Strictly speaking, this only applies to cases when refinement is monotonic. However, all the popular formal methods enjoy this property and heavily rely on it.

Data Refinement: With data refinement, data types are changed and data structures are introduced. The vector of model variables v is changed to some new vector u and model invariant $I(v)$ is replaced with new invariant $J(v, u)$, often called a *gluing invariant*. The use of variables v in new invariant J allows modeller to express a linking relation between the state of concrete and abstract models.

Behavioural Refinement: Behaviour refinement details the mode view on a system. System behaviour becomes more deterministic and also described in a finer level of details. One case is changing a mode assumption or guarantee or both. It is postulated mode assumption cannot be strengthened during refinement. This is based on understanding that an assumption is a requirement of a mode to its environment. As a system developer cannot assume control over the environment of a modelled system, a stronger requirement to an environment may not be realisable. On the other hand, a weaker requirement to an environment means that a system is more robust as it would remain operational in a wider range of environments. Symmetrically, a mode guarantee cannot be weakened as it is understood as a contract of a mode with the rest of a system and its environment. Weakening a mode guarantee could violate expectations of another system part. The following condition summarises this refinement rule:

$$A(v)/G(v, v') \sqsubseteq A'(u)/G'(u, u'), \quad (5)$$

$$\text{iff } \begin{cases} I(v) \wedge J(v, u) \wedge A(v) \Rightarrow A'(u) \\ J(v, u) \wedge G'(u, u') \Rightarrow G(v, v') \end{cases}$$

Another case is when an abstract mode is a modelling abstraction for several concrete modes. Thus, a single mode in an abstract model evolves into a two or more concrete modes. The general rule for such refinement step is that the combination of new modes must be a refinement of an abstract mode. In more concrete terms, a disjunction of concrete mode assumptions must be not stronger than the abstract mode assumption and the disjunction of concrete guarantee must be not weaker than the abstract guarantee:

$$A(v)/G(v, v') \sqsubseteq \begin{matrix} A_1(u)/G_1(u, u') \\ A_2(u)/G_2(u, u') \end{matrix}, \quad (6)$$

$$\text{iff } \begin{cases} I(v) \wedge J(v, u) \wedge A(v) \Rightarrow A_1(u) \vee A_2(u) \\ I(v) \wedge J(v, u) \wedge G_1(u, u') \vee G_2(u, u') \Rightarrow G(v, v') \end{cases}$$

Superposition Refinement: Sometimes it is needed to add new modes without splitting an existing abstract mode. Through superposition refinement it is possible to refine an implicit skip mode *false/true*. This is the weakest form of a mode and it can be refined into any other mode.

Refinement of Transitions: A refinement of a mode or an introduction of a new mode requires changes to mode transitions. The general rule is that a transition present in an abstract model must have a corresponding transition in a refined model and no new transitions may appear. Changing mode assumptions and guarantees does not affect mode transitions. Splitting a mode into sub-modes, however, leads to the distribution of the mode transitions associated with the

refined mode among the new modes. Thus, if a mode with a transition is split into two new modes, the transition can be associated with any one of the new modes or both.

Visual Notation: To assist in application of the approach, a visual notation loosely based on Modechards [6] is proposed. A mode is represented by a box with name; a mode transition is an arrow connecting the previous and next modes. Special modes \top and \perp are omitted so that initiating and terminating transitions appear to be connected with a single mode. Refinement is expressed by nesting boxes. Figure 2 exemplifies this. A transition from an abstract mode is equivalent to having transitions from each of the concrete modes, e.g. transition *ccOff* from abstract mode Cruise Control in diagram (C) of Figure 2.

IV. MODES FOR FAULT TOLERANT SYSTEMS

The use of modes together with a refinement approach, as introduced in the previous sections, offers suitable abstractions to modelling and reasoning about fault tolerant systems, as discussed in the following. Due to the use of a state-based approach, state representation, manipulation and reasoning becomes natural. The support provided by modes allows to partition the state space into normal and erroneous: mode assumptions allow this separation to be declared and erroneous states made explicit. Refinement allows further definition of erroneous states into more specific ones. Assumptions on normal and erroneous states can be suitably associated to modes in charge of performing normal system operation and fault tolerance measures, respectively.

In general, a recovery mode should be associated with a particular normal mode, which it recovers, and mode switching is in some sense reminiscent to calling an exception handler in programming languages. Error detection is immediate, embedded in the erroneous state assumption of a recovery mode. As soon as a state transition leads to the characterization of an erroneous state, the recovery mode is enabled. A more concrete view is to consider the existence of a detection mechanism, which is active during normal operation. In such case the detection mechanism affects the state used in the assumptions of normal and recovery modes. By refinement one could start with the first and reach the second, more detailed model. Any of the possibilities allow switching to recovery mode from any normal mode state. For reasoning purposes, one can introduce the possibility of fault occurrences in parallel with the model. In an event based formalism this takes the form of an enabled event that affects the state to satisfy the erroneous state assumption.

The recovery mode has access to the state of the respective normal mode. Analogously to assumptions, guarantees associated to normal or recovery modes assist to define properties of the system in absence or presence of errors, respectively. Depending on the severity of the detected error, the mode system may assert that the recovery procedure: (i) successfully recovers the state and thus switches back to

normal mode to proceed execution (Figure 1(B) or (C)); (ii) provides degraded service in cases where full functionality is not recoverable (Figure 1(D)); (iii) fails to recover, in which case measures to stop safely may be taken (Figure 1(A) and part of (D)).

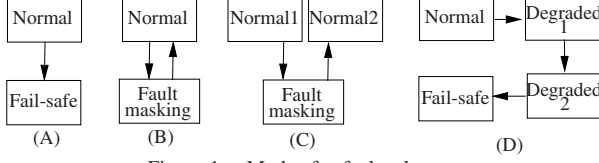


Figure 1. Modes for fault tolerance.

V. OPERATION MODES FOR EVENT-B

The operation modes method is not intended to be used as a modelling method on its own. The schematic nature of the approach makes it well suited to integration with an existing formalism. In this section we discuss how modes can be used with a well known formalism: Event-B. The rules for deriving formal conditions for reasoning about a combination modes and Event-B models are presented.

Event-B is a state-based formalism closely related to Classical B [10] and Action Systems [11]. The step-wise refinement approach is the corner stone of the Event-B development method. The combination of model elaboration, atomicity refinement and data refinement helps to formally transition from high-level architectural models to very detailed, executable specifications ready for code generation.

An extensive tool support through the Rodin Platform makes Event-B especially attractive [12]. An integrated Eclipse-based development environment is actively developed, well-supported, and open to third-party extensions in the form of Eclipse plug-ins. The main verification technique is theorem proving supported by a collection of powerful theorem provers. The development environment is also equipped with model checking capabilities.

An Event-B model is defined by a tuple (c, s, P, v, I, R_I, E) where c and s are constants and sets known in the model; v is a vector of model variables; $P(c, s)$ is a collection of axioms constraining c and s . I is a model invariant limiting the possible states of v : $I(c, s, v)$. The combination of P and I should characterise a non-empty collection of suitable constants, sets and model states: $\exists c, s, v \cdot P(c, s) \wedge I(c, s, v)$. The purpose of an invariant is to express model safety properties (that is, unsafe states may not be reached). In Event-B an invariant is also used to deduce model variable types. R_I is an initialisation action computing initial values for the model variables; it is typically given in the form of a predicate constraining next values of model variables without, however, referring to previous values - $R_I(c, s, v')$. Finally, E is a set of model events. An event is a guarded command:

$$H(c, s, v) \rightarrow S(c, s, v, v') \quad (7)$$

where $H(c, s, v)$ is an event guard and $S(c, s, v, v')$ is a before-after predicate. An event may fire as soon as the condition of its guard is satisfied. In case there is more than one enabled event at a certain state, the demonic choice semantics is applied. The result of an event execution is some new model state v' . The semantics of an Event-B model is usually given in the form of proof semantics, based on Dijkstra's work on weakest precondition [13]. A collection of proof obligations is generated from the definition of the model and these must be discharged in order to demonstrate that the model is correct.

Putting it as a requirement that an enabled event produces a new state v' satisfying a model invariant, the following would define the model *consistency* condition: whenever an event on an initialisation action is attempted there exists a suitable new state v' such that a model invariant is maintained - $I(v')$. This is usually stated as two separate proof obligations: a feasibility obligation requiring the existence of (any) new state v' and the invariant satisfaction obligation showing that any new state v' maintains an invariant. The invariant satisfaction obligation requires that a new state produced by an event must satisfy a model invariant:

$$I(c, s, v) \wedge P(c, s) \wedge H(c, s, v) \wedge S(c, s, v, v') \Rightarrow I(c, s, v') \quad (8)$$

An event must also be feasible: an appropriate new state v' must exist for some given current state v :

$$I(c, s, v) \wedge P(c, s) \wedge H(c, s, v) \Rightarrow \exists v' \cdot S(c, s, v, v') \quad (9)$$

Conceptually, operation modes and Event-B models are related by requiring that every mode and mode transition has a suitable implementation in an Event-B model. A mode is related to a non-empty subset of Event-B model events and mode transition is mapped into a single Event-B event:

$$\begin{array}{l} A_1/G_1 \mapsto E_1, \quad \dots \quad A_n/G_n \mapsto E_n \\ (i \rightsquigarrow j) \mapsto E_p \quad \dots \quad (k \rightsquigarrow l) \mapsto E_q \end{array} \quad (10)$$

Event sets E_1, \dots, E_n may overlap but may not be identical. In the latter case they specify the same mode. The mapping between transitions and events is one-to-many: a transition is mapped into a non-empty set of events. Each event associated with a transition must properly implement the transition, that is, it must be proven it gets enabled in a state assumed by a source mode and establishes a state corresponding to the assumption of a target mode. To establish mapping, for some transition $(i \rightsquigarrow j) \mapsto E_p$ it is required to demonstrate the following:

$$\forall e \cdot (e \in E_p \wedge I(c, s, v) \wedge H_e(c, s, v) \wedge S_e(c, s, v, v') \Rightarrow A_i(v) \wedge A_j(v')) \quad (11)$$

The composition of modes and Event-B clarifies how a system evolves when it is in a mode, how mode switching is done and the way system is initialised. The old *internal* rule is changed to reflect the way a new system state is computed: assuming that a system is mode $A_i/G_i \mapsto E_i$ and the current

state is valid ($I(v)$ holds) and satisfies the mode assumption (A_i holds) the next state is some state v' such that mode guarantee $G(v, v')$ holds along with before-after predicate $R_e(v, v')$ of one of enabled ($H_e(v)$) mode events ($e \in E_i$):

$$\boxed{\text{internal}_1} \frac{I(v) \wedge A_m(v) \wedge G_m(v, v') \wedge A_m(v') \quad \exists e \cdot e \in E_i \wedge H_e(v) \wedge R_e(v, v')}{\langle m, w \rangle \rightarrow \langle m, w' \rangle}$$

The above states that an execution cannot progress if none of the events establishes a mode guarantee or there is no enabled event. To ensure that in a given mode a system evolves correctly, it is required to show for every mode event that the event establishes mode guarantee and the event guard is compatible with the mode assumption. Rules switching_1 and start_1 are analogously obtained from rules switching and start in Section II. The rule above gives a rise to a number of conditions on Event-B. Firstly, all the events of a mode must satisfy its guarantee provided the assumption holds:

$$I(v) \wedge A(v) \wedge H(v) \wedge R(v, v') \Rightarrow G(v, v') \quad (12)$$

Also, the partitioning of the events into modes must be in an agreement with the event guards. When event is enabled then the assumption of its mode must hold. Since an event is potentially associated with multiple modes, the disjunction of all the relevant assumptions must hold:

$$\begin{aligned} H(v) &\Rightarrow A_1(v) \vee \dots \vee A_k(v) \\ A_{k+1}(v) \vee \dots \vee A_n(v) &\Rightarrow \neg H(v) \end{aligned} \quad (13)$$

where A_1, \dots, A_k are the assumptions of the modes containing an event with guard $H(v)$ and A_{k+1}, \dots, A_n are those not containing the event.

It is required to show that a system is always able to progress once it is in a given mode. For this, it must be shown that there is always at least one enabled event among the events of the mode:

$$I(v) \wedge A(v) \Rightarrow H_1(v) \vee \dots \vee H_n(v) \quad (14)$$

Provided the three conditions above are discharged, it is guaranteed that, once in a given mode, a system would unfaillingly progress in accordance with the mode conditions for the system lifetime or until the system transitions into a different mode.

a) Operation Modes and Event-B Co-refinement: The Event-B development method offers a gradual, refinement-based, model detailing. To refine model M one constructs a new model M' such that for any valid state of M' there is a corresponding state in M . In Event-B, this is accomplished by discharging a number of refinement proof obligations formulated for each model event. As refinement in Event-B is monotonic, a model refinement could be constructed by changing only a part of a model and demonstrating the relevant conditions for just that part. Event-B refinement is a combination of data, superposition, behavioural and atomicity refinement. Atomicity refinement permits introduction of a

finer level of atomic steps needed to realise a given functionality. Event-B behavioural refinement allows a modeller to replace an event guard and event before-after predicate. The rules linking abstract and concrete guards and before-after predicates are as follows. The guard of the concrete version of an event must be stronger than its abstract counterpart:

$$P(s, c) \wedge I(s, c, v) \wedge J(s, c, v, u) \wedge H(s, c, u) \Rightarrow G(s, c, v) \quad (15)$$

A new before-after predicate must be a stronger version of its abstraction:

$$\begin{aligned} P(s, c) \wedge I(s, c, v) \wedge J(s, c, v, u) \wedge H(s, c, u) \wedge \\ S(s, c, u, u') \Rightarrow v' \cdot (R(s, c, v, v') \wedge J(s, c, v', u')) \end{aligned} \quad (16)$$

An event may be split into two or more events. In this case, the refinement relation is proved for each new event in the same manner for as for on-to-one event refinement. New events may be introduced but may only update new variables. Standard consistency conditions apply.

A composition of operation modes and Event-B models has to be refined in such a manner that it obeys both operation mode refinement and Event-B refinement. For rule 5, it is required that a refined operation mode is made of events refining events from an abstract mode and also each event from the abstract mode is present as a copy or a refined event in the refined mode.

$$\begin{aligned} A(v)/G(v, v') \mapsto E \sqsubseteq A'(u)/G'(u, u') \mapsto E', \\ \text{iff} \begin{cases} I(v) \wedge J(v, u) \wedge A(v) \Rightarrow A'(u) \\ I(v) \wedge J(v, u) \wedge G'(u, u') \Rightarrow G(v, v') \\ \forall e \cdot e \in E' \Rightarrow \exists a \cdot a \in E \wedge e \sqsubseteq a \\ \forall e \cdot e \in E \Rightarrow \exists a \cdot a \in E' \wedge a \sqsubseteq e \end{cases} \end{aligned} \quad (17)$$

Rule 6 for refinement of modes into a collection of new modes is changed in a similar manner.

$$\begin{aligned} A(v)/G(v, v') \mapsto E \sqsubseteq \begin{matrix} A_1(u)/G_1(u, u') \mapsto E_1 \\ A_2(u)/G_2(u, u') \mapsto E_2 \end{matrix}, \\ \text{iff} \begin{cases} I(v) \wedge J(v, u) \wedge A(v) \Rightarrow A_1(u) \vee A_2(u) \\ I(v) \wedge J(v, u) \wedge G_1(u, u') \vee G_2(u, u') \Rightarrow G(v, v') \\ \forall e \cdot e \in E_1 \cup E_2 \Rightarrow \exists a \cdot a \in E \wedge e \sqsubseteq a \\ \forall e \cdot e \in E \Rightarrow \exists a \cdot a \in E_1 \cup E_2 \wedge a \sqsubseteq e \end{cases} \end{aligned} \quad (18)$$

Conditions 17 and 18 state how mode refinement is related to Event-B refinement. They are the basis for generating proof obligations that would determine the correspondence between an Event-B model and a modes model.

b) Tool Support: The Rodin platform supports modelling and reasoning with Event-B models. Extensions to the Rodin platform can be integrated with: tool interface, modelling process and verification infrastructure. An extension providing the support for modelling with modes would let a designer to visually construct a modes model and would take care of generating the proof obligations required to demonstrate the correspondence between the modes model and the associated Event-B model. Proof obligations are delegated to the proof infrastructure of the Platform that passes them on to one or of automated theorem provers and also an interactive prover should a theorem prover find a problem or fail to discharge a proof obligation.

VI. CRUISE CONTROL CASE STUDY

A simplified version of one of the DEPLOY case studies [14] developed in cooperation with industrial partners, the case study illustrates the application of the proposed technique to the development of a cruise control system. The purpose of the system is to assist a driver in reaching and maintaining a predefined speed. Due to the nature of the system, attention is given to the interaction of a driver, cruise control and the controlled parts of a car. In the current modelling we assume an idealised car and idealised driving conditions such that the car always responds to the commands and the actual speed is updated according to the control system commands. Figure 2 presents the diagrams with the sequence of refinements.

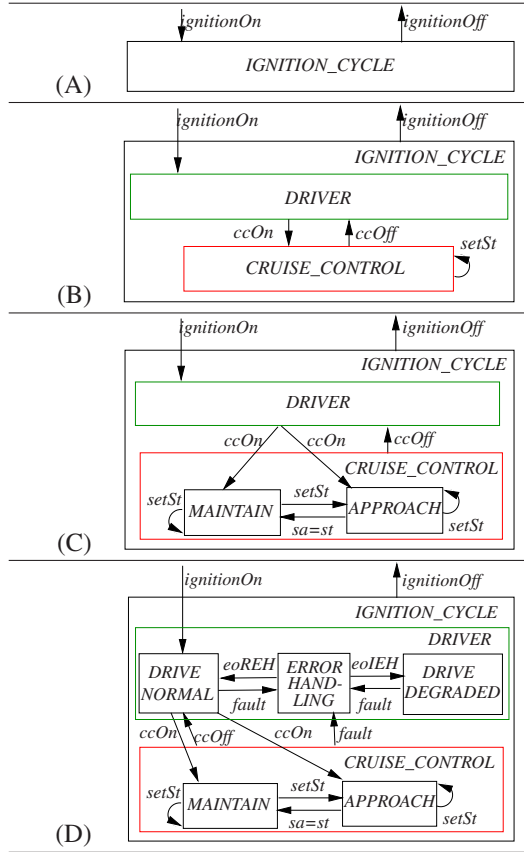


Figure 2. Mode refinement sequence for the Cruise Control System.

(A) *IGNITION_CYCLE*: Figure 2(A) presents the diagram of the most abstract model for the system. It is composed only by the *IGNITION_CYCLE* mode and represents the activity from the instant the ignition is turned on, event *ignitionOn* establishes the assumption for that mode, to the instant it is turned off, event *ignitionOff* changes the conditions of the system and falsify the assumptions for this mode. The model includes: the state of ignition (on/off), modelled by a boolean flag *ig*; the current speed of the car (a modelling approximation of an actual car speed), stored

in variable *sa*; a safe speed limit *speedLimit* above which the car should not be in any case; and a safe speed variation *maxSpeedV*. No memory is retained about the states in the previous ignition cycle. Initially, the current speed is zero and ignition is off: $sa \in 0 \wedge ig \in FALSE$. Independently of the operation of the car (by the driver or by the cruise control) the following has to be ensured during an ignition cycle (we present the intuition in the first line and a formal representation of the assumptions and guarantees, based on the variables introduced, in the second line).

mode	assumption	guarantee
<i>IGNITION_CYCLE</i>	ignition is on	keep speed under limit and (ac/de)celerate safely
	$ig = true$	$(sa < speedLimit) \wedge (sa' - sa < maxSpeedV)$

(B) *DRIVER* and *CRUISE_CONTROL*: When the ignition is turned on, control is with the driver. While the ignition is on, control can be passed from the driver to the cruise control and back. It is assumed that a driver has two buttons on a control panel: the *on* button switches on the cruise control; the *off* button returns to the driving mode. A third input is available to set the target speed to be achieved by the cruise control. The system is naturally represented with two modes: *DRIVER* corresponding to the activity when cruise control is off and *CRUISE_CONTROL* when cruise control is active. The *on/off* buttons mentioned are mapped to transition events *ccOn* and *ccOff*. The diagram in Figure 2(B) depicts the two possible modes during an ignition cycle.

This refinement introduces: the state of cruise control (on/off), modelled by boolean flag *cc*; the target speed that a cruise control is to achieve and maintain, represented by variable *st*; an allowance interval *isp* that determines how much actual speed could deviate from a target speed when cruise control tries to maintain a target speed. Initially, the target speed is undefined and cruise control is off: $st \in \mathbb{N} \wedge cc \in FALSE$. The description of the modes:

mode	assumption	guarantee
<i>DRIVER</i>	ignition cycle assumptions and cruise control off	ignition cycle guarantees
	$ig = true \wedge cc = false$	$(sa < speedLimit) \wedge (sa' - sa < maxSpeedV)$
<i>CRUISE_CONTROL</i>	ignition cycle assumptions and cruise control on	ignition cycle guarantees and maintain target speed or approach target speed
	$ig = true \wedge cc = true$	$(sa < speedLimit) \wedge (sa' - sa < maxSpeedV) \wedge (sa' - st' \leq isp \vee sa' - st' < sa - st)$

(C) *Refining the CRUISE_CONTROL Mode*: If the difference between current (*sa*) and target (*st*) speeds is within an acceptable error interval (*isp*), the cruise control works to *MAINTAIN* the current speed. Otherwise, it employs different procedures to *APPROACH* the target speed, characterizing two modes refining *CRUISE_CONTROL*. Respective

assumptions and guarantees are described in the table below. Figure 2(C) depicts these modes. Switching from *DRIVER* to *CRUISE_CONTROL* may either establish the assumptions of *APPROACH* or *MAINTAIN*, depending on the difference between st and sa . In either of these modes the cruise control can be switched off and control returned to the driver.

mode	assumption	guarantee
<i>APPROACH</i>	cruise control assumptions and speed not close to target	cruise control guarantees and approach target speed
	$ig = true \wedge cc = true \wedge sa' - st' > isp$	$(sa < speedLimit) \wedge (sa' - sa < maxSpeedV) \wedge (sa' - st' < sa - st)$
<i>MAINTAIN</i>	cruise control assumptions and speed close to target	cruise control guarantees and maintain target speed
	$ig = true \wedge cc = true \wedge sa' - st' \leq isp$	$(sa < speedLimit) \wedge (sa' - sa < maxSpeedV) \wedge (sa' - st' \leq isp)$

(D) *Error handling*: at any time failures of the surrounding components (e.g. airbag activated, low energy in battery) may affect the cruise control system. These faults are signaled as erroneous conditions and can be either reversible or irreversible: the reversible errors result in the control to be returned to the driver and handling measures to be undertaken, so that the cruise control becomes available again; the irreversible ones are handled but the cruise control becomes unavailable during the ignition cycle.

mode	assumption	guarantee
<i>DRIVE_NORMAL</i>	driver assumptions and no error	driver guarantees and cruise control available
	$ig = true \wedge cc = false \wedge error = false$	$(sa < speedLimit) \wedge (sa' - sa < maxSpeedV)$
<i>ERROR_HANDLING</i>	driver assumptions and error and handling not finished	driver guarantees and cruise control not available and recovery measures restore normal mode or switch to degraded mode
	$ig = true \wedge cc = false \wedge error = true \wedge eHand = true$	$(sa < speedLimit) \wedge (sa' - sa < maxSpeedV)$
<i>DRIVE_DEGRADED</i>	driver assumptions and error and handling finished	driver guarantees and cruise control not available
	$ig = true \wedge cc = false \wedge error = true \wedge eHand = false$	$(sa < speedLimit) \wedge (sa' - sa < maxSpeedV) \wedge$

When an error is detected it is registered in an *error* variable. We introduce a normal (*DRIVE_NORMAL*), a de-

graded (*DRIVE_DEGRADED*) and an error handling mode (*ERROR_HANDLING*). If an error is signaled in any of the system modes, the system switches to *ERROR_HANDLING*, where control is with the driver. Eventually error handling reestablishes *DRIVE_NORMAL*, with full functionality available, or switches to *DRIVE_DEGRADED* mode where the cruise control is not available. This exemplifies situations (C) and (D) of Figure 1. Figure 2(D) shows these modes. An *eHand* variable registers that error handling is taking place. The following table shows the assume/guarantee conditions for the modes introduced. Note that although these modes have same guarantees, they have different transition possibilities. After error handling, the system continues in degraded or normal mode. From error handling and degraded modes it is not possible to turn the cruise control on.

VII. RELATED WORK

Several applications, structured in modes, can be found in the literature. Papers [7] and [8] show how to formally model and analyse modal space and avionic systems. In [15] an Automated Highway System is extended to tolerate several kinds of faults, modes are used to characterize degraded operation. A classic case study on formal methods, the Steam Boiler Control [16], is based on operation modes. More recent examples on the use of modes for the specification of airspace, transportation and automotive systems can be found in [14]. Such contributions focus on specific applications and not on general means to model and reason using modes.

In [17] the authors discuss characteristics of mode-driven distributed applications and an infrastructure is proposed to support mode-driven fault tolerance in run time. In [18], the representation of degraded service outcomes and exceptional modes of operation using UML use cases, activity diagrams and state charts is discussed. Formal modelling and reasoning is not discussed in these contributions.

In [6] a specification language for real-time systems, called Modechart, is presented. In [19] the author discusses issues related to mode changes and scheduling for hard real-time systems. The general notion of modes in these papers is analogous to the one discussed here, however their focus is on the specification and analysis of timing properties of systems. Functional properties are not discussed.

In the context of refinement based methods, the most related work found is by Back and von Wright [20], where guarantees (of an action system) are introduced to reason about the parallel composition of action systems. Guarantees of composed action systems have to mutually respect the invariants. Since there is no notion of assumptions, the flexibility of allowing different modes and mode switching, is not offered.

Finally, Jones, Hayes and Jackson, in [21], discuss a method that leads the designer to explicitly state rely conditions (to be compared with assumptions) about the physical world before deriving a first specification of the system. The

notion of 'layer' is briefly discussed. A layer is associated to a set of rely/guarantee predicates and could be compared to a mode. Different layers could be used to state the behaviour under distinct conditions. Fault tolerance is briefly mentioned, where one could have assumptions to characterise absence or presence of faults.

VIII. CONCLUSIONS

In this paper the notions of modes and mode refinement are formally defined and their representations in a state-base formalism (Event-B) are established. These notions allow explicit characterization of various system conditions, through expressing assumptions, and the properties of the system working under such conditions, through the use of guarantees. The complexity of design is reduced by structuring systems using modes and by detailing this design using refinement. This approach makes it easier for the developers to map requirements to models and to trace requirements. More specifically, the approach suits well for dealing with fault-tolerance requirements: assumptions allow the explicit mapping of the error coverage provided by the system, whereas guarantees and mode switching configurations allow the explicit mapping of requirements for different levels of fault-tolerance.

In addition to developing a tool support, in the near future we plan to investigate mode hierarchy (nesting), to express recursive structuring for fault tolerance [22], mode concurrency, where further work is needed to support concurrent modes acting on shared states, and state consistency during distributed execution of modes.

REFERENCES

- [1] J.-C. Laprie, B. Randell, A. Avizienis, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, 2004.
- [2] F. Cristian, *Exception handling*, T. Anderson, Ed. Blackwell Scientific Publications, 1989.
- [3] A. Romanovsky, "A looming fault tolerance software crisis?" *SIGSOFT Softw. Eng. Notes*, vol. 32, no. 2, pp. 1–4, 2007.
- [4] J. Peleska, "Formal methods and the development of dependable systems - habilitationsschrift," Institut für Informatik und Praktische Mathematik der Christian-Albrechts-Universität zu Kiel, Tech. Rep. 9612, 1996.
- [5] F. C. Gärtner, "Transformational approaches to the specification and verification of fault-tolerant systems: formal background and classification," *Journal of Univ. Computer Science*, vol. 5, no. 10, pp. 668–692, 1999.
- [6] F. Jahanian and A. Mok, "Modechart: A specification language for real-time systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 933–947, 1994.
- [7] R. W. Butler, "An introduction to requirements capture using pvs: Specification of a simple autopilot," NASA, Tech. Rep. 110225, 1996.
- [8] S. P. Miller, "Specifying the mode logic of a flight guidance system in core and scr," in *FMSP '98: Proc. of the 2nd workshop on Formal methods in software practice*. New York, USA: ACM, 1998, pp. 44–53.
- [9] R.-J. J. Back and J. V. Wright, *Refinement Calculus: A Systematic Introduction*. Springer NY, Inc., 1998.
- [10] J. R. Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.
- [11] R.-J. Back and K. Sere, "Stepwise Refinement of Action Systems," in *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniv. of the Groningen Univ.*, J. L. A. van de Snepscheut, Ed. London, UK: Springer-Verlag, 1989, pp. 115–138.
- [12] "Event-b and the rodin platform," <http://www.event-b.org/> (last accessed 8 March 2009). Rodin Development is supported by European Union ICT Projects DEPLOY (2008 to 2012) and RODIN (2004 to 2007).
- [13] E. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.
- [14] J.-R. Abrial, J. Bryans, M. Butler, J. Falampin, T. S. Hoang, D. Ilic, T. Latvala, C. Rossa, A. Roth, and K. Varpaaniemi, "Report on knowledge transfer - deploy deliverable d5," p. 321, February 2009.
- [15] J. Lygeros, D. N. Godbole, and M. E. Broucke, "Design of an extended architecture for degraded modes of operation of ivhs," in *In American Control Conf.*, 1995, pp. 3592–3596.
- [16] J.-R. Abrial, E. Börger, and H. Langmaack, Eds., *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*, ser. Lecture Notes in Computer Science, vol. 1165. Springer, 1996.
- [17] D. Srivastava and P. Narasimhan, "Architectural support for mode-driven fault tolerance in distributed applications," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–7, 2005.
- [18] S. Mustafiz, J. Kienzle, and A. Berlizev, "Addressing degraded service outcomes and exceptional modes of operation in behavioural models," in *SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*. New York, NY, USA: ACM, 2008, pp. 19–28.
- [19] G. Fohler, "Realizing changes of operational modes with a pre run-time scheduled hard real-time system," in *In Proceedings of the Second International Workshop on Responsive Computer Systems*. Springer Verlag, 1992, pp. 287–300.
- [20] R.-J. Back and J. von Wright, "Compositional action system refinement," *Formal Asp. Comput.*, vol. 15, no. 2-3, pp. 103–117, 2003.
- [21] C. B. Jones, I. J. Hayes, and M. A. Jackson, "Deriving specifications for systems that are connected to the physical world," in *Formal Methods and Hybrid Real-Time Systems*, 2007, pp. 364–390.
- [22] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*, J. C. Laprie, A. Avizienis, and H. Kopetz, Eds. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1990.