Structuring Specifications with Modes

Alexei Iliasov
Newcastle University
Email: alexei.iliasov@newcastle.ac.uk

Fernando L. Dotti
Newcastle University/PUCRS-Brazil
Email: fernando.dotti@pucrs.br

Alexander Romanovsky
Newcastle University
Email: alexander.romanovsky@newcastle.ac.uk

***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.**

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].

Several authors have investigated fault-tolerance modelling using different specification formalisms and verification approaches (see surveys [4], [5]). However, the identification and support of suitable abstractions for formal design of fault tolerant systems is still an open area. 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, and state manipulation to reach consistency are 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 fault tolerance mechanisms; and (iii) system properties under changing conditions - the explicit statement of possible working conditions, addressing fault assumptions, and assured system properties under these conditions are also to be supported.

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. This is a REGULAR PAPER.

In this paper the known concept of 'operation mode' [6] is revisited - we use modes to structure system specification to facilitate rigorous design and to integrate fault tolerance. Due to the use of modes in different types of systems, such as real-time [6], avionic and space systems [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, as well as a way to define control information in large state machines, imposing structure on the operation of the system. The modes are defined to allow the modeller to explicitly state the property that must be respected, called guarantee, in each working system condition, called assumption. These notions are discussed in Section II.

Modes can be refined, allowing detalisation of the system (see Section III), and can be used together with a state-based formal method (see Section V). Mode refinement is performed hand in hand with the refinement of the respective formal model and offer a way for layered definition and reasoning about system properties. This helps to assure that the properties are easily traceable to requirements. The case study in Section VI exemplifies this. Another advantage of a refinement approach is that it offers a strategy to obtain a correct implementation from the formal model. Moreover, theorem proving strategies and tools sometimes offer an attractive option to model-checking as they avoid the state-space problem.

This paper is organized as follows: modes and mode refinement are presented in Sections II and III; Section IV discusses their use in the design of fault-tolerant systems; Section V discusses the use of modes with the Event-B formalism; the model of a Cruise Control system is presented in Section VI; related work and conclusions make Sections VII and VIII.

II. Operation Modes

Operation modes help to reason about the behaviour of complex systems by focusing on the principal 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 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; and

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 of v 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.

$$I(v) = A_1(v) \oplus \cdots \oplus A_n(v) \tag{1}$$

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 $t \rightsquigarrow \bot$ [1]. Mode transitions $i \rightsquigarrow \top$ and $\bot \rightsquigarrow j$ are not allowed. Also, it is required that during its lifetime a system enters at least in one operation mode and thus transition $\top \rightsquigarrow \bot$ is not possible. There can be any number of initiating and terminating mode transitions.

There are certain restrictions on the way mode assumptions and guarantees are formulated. One obvious condition is that a guarantee may not require or permit a mode to violate an invariant, that is, the states described by a guarantee must be wholly included into valid model states:

$$I(v) \wedge A(v) \wedge G(v, v') \Rightarrow I(v') \tag{2}$$

The assumption and guarantee of a mode must be non-contradictory. I.e. a mode should permit a concrete implementation[2]:

$$\exists v, v' \cdot (I(v) \wedge A(v) \Rightarrow G(v, v')) \tag{3}$$

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

[2]An example of malformed mode is $x \in NAT/x' * x' < x$.

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

$$\begin{array}{c} A_1/G_1 \\ \cdots \\ A_n/G_n \\ i \rightsquigarrow j \\ \cdots \end{array} \tag{4}$$

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. There are two types of non-deterministic choice: one when it is resolved by a system itself; the other relies on an entity beyond system boundaries. In this approach, the latter case is applied. An important implication is that the only way to ensure proper functioning of a system in spite of unfavourable behaviour of an environment is to sufficiently constrain model non-determinism. An initiating transition is a special case: it must find an initial system state without being able to refer to any previous state:

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

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: $I(w) \wedge I(w')$. This is a corrolary of conditions 1 and 3.

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 behaviour[3].

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

At a very general level, 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 completely formal in a sense that a refined model may be used in place of its abstraction.

Refinement itself is a combination of a number of techniques: data refinement, when data types are changed and data structures are introduced; behavioural refinement, when system behaviour becomes more deterministic and also described in a finer level of details; and superposition refinement (or model elaboration), when new functionality is added without changing an existing model. All the three are applicable and discussed for modes in the following.

a) Data Refinement: With data refinement, 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 mentioning of old variables v in new invariant J allows modeller to expresses a linking relation between the state of concrete and abstract models.

b) Behavioural Refinement: Behaviour refinement details the mode view on a system. 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 a mode guarantee is understood as a contract of a mode with the rest of a system and the system environment. In other words, weakening a mode guarantee could violate expectations of another system part. The following condition summarises this refinement rule:

$$\begin{aligned} &A(v)/G(v, v') \sqsubseteq A'(u)/G'(u, u'), \\ &\text{iff } \begin{cases} I(v) \wedge J(v, u) \wedge A(v) \Rightarrow A'(u) \\ J(v, u) \wedge G'(u, u') \Rightarrow G(v, v') \end{cases} \end{aligned} \tag{5}$$

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:

$$\begin{aligned} &A(v)/G(v, v') \sqsubseteq \begin{array}{l} A_1(u)/G_1(u, u') \\ A_2(u)/G_2(u, u') \end{array}, \\ &\text{iff } \begin{cases} I(v) \wedge J(v, u) \wedge A(v) \Rightarrow A_1(u) \vee A_2(u) \\ I(v) \wedge J(v, u) \wedge G_1(u, u') \vee G_2(u, u') \Rightarrow G(v, v') \end{cases} \end{aligned} \tag{6}$$

c) Superposition Refinement: Sometimes it is needed to add new modes without having to split an abstract mode. This is accomplished using superposition refinement. With superposition refinement, a refined model contains additional modes. Assumptions and guarantees of these modes must be expressed on new variables (variables for u that are not mapped onto abstract variables v). Formally, this is possible by refining an implicit skip mode $false/true$. This is the weakest form of a mode and it can be refined into any other mode.

d) 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.

e) Visual Notation: To assist in application of the operation modes approach, a simple visual notation is proposed. It is loosely based on modechards [6]. A mode is represented by a box with mode name; a mode transition is an arrow connecting two modes. The direction of an arrow indicates the previous and next modes in a transition. Special modes $\top$ and $\bot$ are omitted in a diagram so that initiating and terminating transitions appear to be connected with a single mode. This is also how they can be distinguished from other transitions. Refinement is expressed by nesting boxes. Figure 1(B) presents a mode M1 refined into modes M1.1 and M1.2. The mode transitions depicted are only one possibility. A refined diagram with an outgoing arrow from an abstract mode is equivalent to having outgoing arrows from each of the concrete modes (this feature is used in the case study).

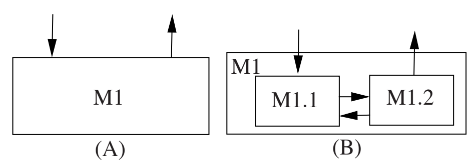

Fig. 1. Mode Diagrams.

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.

Generally speaking, 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 recovery mode guarantees may assert that the recovery procedure: (i) successfully recovers the state and thus switches back to normal mode to proceed execution (Figure 2(B) or (C)); (ii) provides degraded service in cases where full functionality is not recoverable (Figure 2(D)); (iii) fails to recover, in which case measures to stop safely may be taken (Figure 2(A) and part of (D)). Using the graphical notation introduced in the previous section, the following configurations exemplify some possible use of modes for fault tolerance.

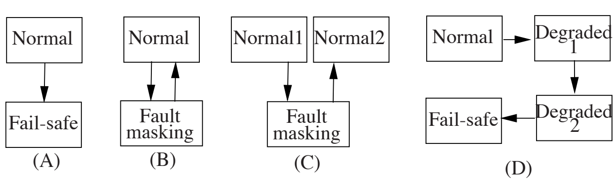

Fig. 2. Modes for fault tolerance.

V. Operation Modes for Event-B

The operation modes method is not intended to be used as a modelling method on its own as it lacks the facilities for expressing detailed design. The schematic nature of the approach makes it it well suited to integration with an existing formalism. One such case is presented in this section. A well known formalism - Event-B - is extended with operation modes. 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:

$$H(c, s, v) \rightarrow S(c, s, v, v') \tag{7}$$

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 and no other event executes at the same time. In case there is more than one enabled event at a certain state, the demonic choice semantics is applies. 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 satisfies a model invariant:

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

An event must also be feasible, in a sense that an appropriate new state v' must exist for some given current state v:

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

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:

$$\begin{array}{l} A_1/G_1 \mapsto E_1 \\ \quad \dots \\ A_n/G_n \mapsto E_n \\ (i \leadsto j) \mapsto E_k \\ \quad \dots \end{array} \tag{10}$$

Event sets $E_1, \dots, E_n$ may overlap but may not be identical. The latter is due to the fact that two modes $A_i/G_i \mapsto E$ and $A_j/G_j \mapsto E$ are equivalent to a single mode $A_i \vee A_i/G_i \wedge G_j \mapsto E$ and thus there is no advantage in allowing configurations where modes have identical event sets. 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 stated assumed by a source mode and establishes a state corresponding to the assumption of a target mode. To establish mapping, for some transition $(i \leadsto j) \mapsto E_k$ it is required to demonstrate the following:

$$\forall e \cdot (e \in E_k \wedge I(c,s,v) \wedge H_e(c,s,v) \wedge S_e(c,s,v,v') \Rightarrow A_i(v) \wedge A_j(v')) \tag{11}$$

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 ($e \in E_i$):

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

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 $\mathsf{switching}_1$ and start_1 are analogously obtained from rules $\mathsf{switching}$ 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 the mode guarantee provided the mode assumption holds:

$$I(v) \wedge A(v) \wedge H(v) \wedge R(v,v') \Rightarrow G(v,v') \tag{12}$$

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:

$$\begin{array}{l} H(v) \Rightarrow A_1(v) \vee \dots \vee A_k(v) \\ A_{k+1}(v) \vee \dots \vee A_n(v) \Rightarrow \neg H(v) \end{array} \tag{13}$$

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:

$$I(v) \wedge A(v) \Rightarrow H_1(v) \vee \dots \vee H_n(v) \tag{14}$$

Provided the three conditions above are discharged, it is guaranteed that, once in a given mode, a system would unfailingly 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 cornerstone of the Event-B development method is a gradual, refinement-based, model detailing. To refine model M one constructs a new model M' such that at a certain level of observation new model is at least as good as the old one. Formally, this is demonstrated by constructing a refinement mapping between M' and M that would show 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. What would appear as an one atomic event in an abstract model may be refined into a complex of events with all the properties of the abstraction retained. The Event-B notion of data refinement follows the same generic style used for operation modes data refinement.

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:

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

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

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

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.

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

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

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

Conditions 17 and 18 state how mode refinement is related to Event-B model 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 for Modes Modelling: As already mentioned, 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 (defined by relation (10)). 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 some predefined speed. Due to the nature of the system, a lot of 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.

a) A Mode for the Ignition Cycle: At the most abstract level we introduce mode *IGNITION_CYCLE* to represent the activity from the instant the ignition is turned on to the instant it is turned off. The initial 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 $maxSpeedVariation$. 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 same assumptions and guarantees, based on the variables introduced, in the second line).

mode	assumption	guarantee
IGNITION_CYCLE	ignition is on	keep speed under limit and (ac/de)celarate safely
	$ig = true$	$(sa < speedLimit) \wedge (\ \lvert sa' - sa \rvert < maxSpeedVariation\ )$

Figure 3 presents the diagram of the system. At this level of abstraction it is composed only by the *IGNITION_CYCLE* mode. An event happens in the system that establishes the assumption for that mode: $ignitionOn$. While ignition is on, the corresponding guarantees have to be ensured. Another event may change the conditions of the system and the assumptions for this mode may become false: $ignitionOff$.

Fig. 3. Abstract model diagram: the ignition cycle.

b) DRIVER and CRUISE_CONTROL Modes: 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 4 depicts the two possible modes during an ignition cycle.

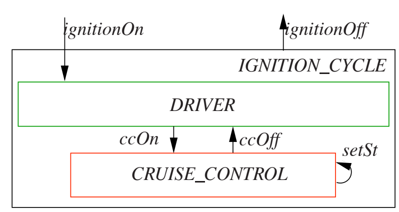

Fig. 4. Introduction of Driver and Cruise Control modes

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
DRIVER	ignition cycle assumptions and cruise control off	ignition cycle guarantees
	$ig = true$ $\wedge$ $cc = false$	$(sa < speedLimit) \wedge (\lvert sa' - sa\rvert < maxSpeedVariation)$
CRUISE_CONTROL	ignition cycle assumptions and cruise control on	ignition cycle guarantees and maintain target speed or approach target speed
	$ig = true$ $\wedge$ $cc = true$	$(sa < speedLimit) \wedge (\lvert sa' - sa\rvert < maxSpeedVariation) \wedge (\lvert sa' - st'\rvert \leq isp \ \vee \ \lvert sa' - st'\rvert < \lvert sa - st\rvert)$

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* with assumptions and guarantees are as follows.

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

Figure 5 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 two modes the cruise control can be switched off and the control returned to the driver.

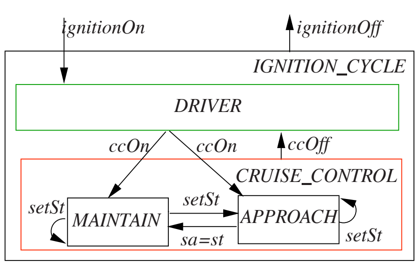

Fig. 5. Introduction of Internal Cruise Control modes

d) Fault handling: at any time failures in other components (e.g. airbag activated, low energy in battery, etc.) may happen and affect the cruise control system, being considered faults for the latter. These faults are signaled to the system and can be reversible or irreversible: reversible faults cause the control to be returned to the driver, handling measures to be undertaken, and the cruise control becomes available again; irreversible faults are handled and the cruise control becomes unavailable during the ignition cycle.

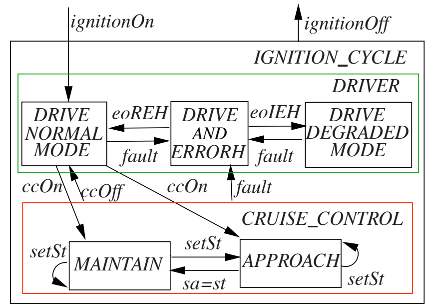

Fig. 6. Introduction of Fault Handling and Degraded modes

The signaling of fault situations is registered in an $error$ variable. We introduce a normal (*DRIVE_NORMAL_MODE*), a degraded service (*DRIVE_DEGRADED_MODE*) and an error handling mode (*DRIVE_AND_ERRORH*). While in any mode of the system a fault may be signaled, switching the system to *DRIVE_AND_ERRORH* where control is with the driver. Eventually the error handling reestablishes *DRIVE_NORMAL_MODE*, 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 2. Figure 6 shows these modes. An $eHand$ variable is used to register that error handling is taking place. The following is a representation based on the variables introduced.

mode	assumption	guarantee
DRIVE_NORMAL_MODE	driver assumptions and no error	driver guarantees and cruise control available
	$ig = true\wedge$ $cc = false\wedge$ $error = false$	$(sa < speedLimit) \wedge (\lvert sa' - sa\rvert < maxSpeedVariation) \wedge (cc' \in BOOL)$
DRIVE_AND_ERRORH	driver assumptions and error and handling not finished	driver guarantees and cruise control not available and recovery measures restore normal mode or swich to degraded mode
	$ig = true\wedge$ $cc = false\wedge$ $error = true\wedge$ $eHand = true$	$(sa < speedLimit) \wedge (\lvert sa' - sa\rvert < maxSpeedVariation) \wedge (cc' \in FALSE) \wedge eHand', error' \in \{(true, true), (false, BOOL)\}$
DRIVE_DEGRA-DED_MODE	driver assumptions and error and handling finished	driver guarantees and cruise control not available
	$ig = true\wedge$ $cc = false\wedge$ $error = true\wedge$ $eHand = false$	$(sa < speedLimit) \wedge (\lvert sa' - sa\rvert < maxSpeedVariation) \wedge (cc' \in FALSE)$

VII. Related Work

Several applications, structured in modes, can be found in the literature. In [7], [8] efforts were invested to the formal modelling and analysis of space and avionic systems, respectively. In [15] the extension of an Automated Highway System to tolerate several kinds of faults is discussed, and modes are used to characterize degraded operation. Such contributions focus on specific applications and not on general means to model and reason using modes.

In [16] the authors discuss characteristics of mode-driven distributed applications and a software architecture with extensions to mode-driven fault-tolerance. An infrastructure is proposed to support mode-driven fault tolerance in run time. In [17], 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. The notion of modes is closely related to the one discussed in this paper, however [6] is focused 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 [18], 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 (they are embedded in the invariants), the flexibility of changing assumptions, allowing different modes and mode switching, is not offered.

Finally, Jones, Hayes and Jackson, in [19], address the problem of obtaining a starting specification for a class of systems, namely those that describe behaviour in the physical world, like control systems. A method is discussed 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, where one layer would be 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 characterize absence of component failures 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 concurrency and mode hierarchy (nesting). The former needs further work to support concurrent modes acting on shared states. The latter should allow us to express recursive structuring for fault tolerance [20]. Another important issue we intend to address is 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," 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, "Nasa tech. memo. 110255 an introduction to requirements capture using pvs: Specification of a simple autopilot," 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-Verlag New York, 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 Int., 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, "Deploy deliverable d5 1dj1 (draft) - report on knowledge transfer," p. 321, Jan. 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 Conference*, 1995, pp. 3592–3596.

[16] 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.

[17] 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.

[18] R.-J. Back and J. von Wright, "Compositional action system refinement," *Formal Asp. Comput.*, vol. 15, no. 2-3, pp. 103–117, 2003.

[19] 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.

[20] 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.