Invariant Management in the Presence of Failures

Richard Banach[1]

[1]School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
Email: banach@cs.man.ac.uk

Abstract. In the effort to develop critical systems, taking account of failure modes is of vital importance. However, when systems fail (even in a manner previously determined as acceptable), a lot of the invariants that hold in the case of nominal behaviour also fail. A technique is proposed that permits the inclusion of the strong invariants of nominal behaviour alongside the provisions for degraded behaviour in an inclusive formal system model. The faulty system model is derived from the nominal one via fault injection, and the nominal and faulty system models are related via a formal retrenchment step. Manipulation of the retrenchment data permits the inclusion of the stronger invariants, which remain provable when faults are disabled in a generic manner in the faulty model, thus increasing confidence in the overall system design. The details are developed in Event-B, and the concept is illustrated using a toy switching example.

1 Introduction

When developing critical systems, it is rarely (in fact never) possible to assume all components will work perfectly all the time. Provision for the judicious handling of degraded operation is a vital concern in the design of all categories of critical system. This creates a dilemma of the following kind. When all components are working normally (nominal behaviour), very many specific invariants will hold about the detailed working of each of the components and about their interworking. The verification of these in the nominal system can provide a useful measure of confidence in the correctness of the model of the desired system. But the system cannot be viewed as being always nominal, so all these detailed invariants cannot be assumed to hold in the real system, when degraded operation in the presence of failure modes is contemplated. So the purported invariants are not in fact invariants, and the confidence in the correctness of the model of the desired system that their verification can provide, is lost. Only fewer, and inevitably weaker invariants will hold in the full system, these capturing the perforce weaker requirements that are demanded of the full system when all foreseen failure modes are taken into account.

In this paper, we present an approach that can straddle these two extremes. Briefly, a suitable formal model of the nominal system is first developed. Since faults are not contemplated at this stage,[1] a suite of incisive invariants is developed accompanying the model. These tightly police the detailed inner workings of the early nominal model,

[1] Of course, in reality, the fault portfolio is contemplated from the earliest stages of development, but we do not include any failure modes in the early stages of formal modelling.

and their verification gives a lot of confidence in the correctness of its design. In the next stage, faults are introduced into the nominal model via fault injection. Of course, this destroys the validity of the strong invariants developed earlier. So the modified model removes those, and contains only invariants sustainable in both nominal and faulty regimes, i.e. weaker ones capturing the requirements of the full system. This process is formalised within a retrenchment development step [13, 12, 11]. But, equally evidently, provided no fault occurs during a run of the system, the earlier stronger invariants will hold true. So, in the next stage, the stronger invariants are reinstated, and the non-occurrence of faults is axiomatised (typically by setting all fault variables permanently to false, or by disabling fault occurrence in some other way) — the formality of the retrenchment step helps to do this in a structured way that is intended to be non-invasive, as regards the pivotal features of the system model. In this model, modified from the faulty system model in a stylised manner, despite the presence of all the fault machinery (except for the actual invocation of faults), the stronger invariants are provable, and their verification gives added confidence in the design, even when the fault machinery is present. That, in a nutshell, is the proposal of this paper.

The rest of this paper is as follows. Section 2 recalls the small set of Event-B [1] and its refinement and retrenchment theory to enable us to handle the concepts of this paper and of our illustrative example. Section 3 introduces a toy switching example in nominal form. Section 4 considers injecting faults into the nominal model, and structures the process using the retrenchment machinery. Section 5 discuses how the stronger invariants of the nominal model can be retained within the failing model using the formal machinery introduced earlier. Section 6 covers related work. Section 7 concludes.

2 Event-B Essentials, Refinement, Retrenchment

In this section we recall a few essential features of Event-B, omitting a large number of facts not needed for our exposition. See [1, 25, 24, 2, 26, 17, 3] for a fuller exposition.

Event-B is a formalism for defining, refining and reasoning about discrete event systems. Its relatively uncluttered design makes it useful in many kinds of application. The syntactic unit that expresses self-contained behaviour is the MACHINE. This declares the VARIABLES of the machine, and crucially, the INVARIANTS that all runs of the machine must conform to. Runs are specified implicitly via successions of EVENTS, each being of the form:

$$\mathit{EvName} \;\widehat{=}\; \text{WHEN}\ \mathit{grd}\ \text{THEN}\ xs := es\ \text{END}$$

In this, grd is a guard, a boolean expression in the variables and constants, the truth of which enables the event to execute (otherwise the event cannot run), and the THEN clause defines a set of parallel updates $xs := es$ to the variables, executed atomically. Of course, there are many additional forms of event syntax in the more definitive [1]. Some of these we will meet below.

For machine M to be *correct*, the following PO schemas must be provable:

$$Init(u') \Rightarrow Inv(u')$$
$$Inv(u) \wedge grd_{Ev}(u) \Rightarrow \exists u' \bullet BApred_{Ev}(u, u')$$
$$Inv(u) \wedge grd_{Ev}(u) \wedge BApred_{Ev}(u, u') \Rightarrow Inv(u')$$

In this, *Init*ialisation is treated as an event so its after-value must establish the machine invariants Inv; grd_{Ev} is the guard of event Ev, and $BApred_{Ev}$ is the before-after predicate of event Ev, specifying in a logical form the update to variables that it defines, with primes referring to after-values. Thus the first PO ensures that $Init$ establishes the invariants, the second PO gives event feasibility (i.e. there is some after-state for an enabled event), and the third PO ensures all event executions maintain the invariants.

Event-B refinement is based on the action refinement model [6, 4, 5, 7]. Suppose a machine MR, with variables v refines machine M with variables u. Suppose the u and v state spaces are related by a refinement invariant $R(u, v)$ (also referred to as the joint invariant). Suppose abstract Ev_A is refined to concrete Ev_C. Then the principal refinement PO schemas are:

$$Init_C(v') \Rightarrow \exists u' \bullet Init_A(u') \wedge R(u', v')$$
$$R(u, v) \wedge grd_{Ev_C}(v) \Rightarrow grd_{Ev_A}(u)$$
$$R(u, v) \wedge grd_{Ev_C}(v) \wedge BApred_{Ev_C}(v, v') \Rightarrow \exists u' \bullet BApred_{Ev_A}(u, u') \wedge R(u', v')$$

In this, the first PO establishes refinement of initialisation, the second is guard strengthening, and the third establishes the simulation property of any concrete step by some abstract step. In addition to refinements of abstract events, the concrete machine may contain new events. These satisfy a PO in which there is no change of the abstract state:

$$R(u, v) \wedge grd_{NewEv_C}(v) \wedge BApred_{NewEv_C}(v, v') \Rightarrow R(u, v')$$

If all of the above are provable for a pair of machines M and MR, then an inductive proof of simulation of any concrete execution by some abstract execution follows relatively readily.

Retrenchment is a looser variant of the refinement relationship between machines. If machine MR with variables v is retrenched to machine MRF with variables w, then firstly, either machine may contain events unrelated by retrenchment to events in the other machine. Secondly, for a pair of corresponding events Ev_A and Ev_C which *are* related by retrenchment, the retrenchment relationship is given by (in addition to the preceding data for refinement), an OUTput clause O_{Ev} and a CONCedes clause C_{Ev}.

The concrete syntax will be exemplified below in our example, but the PO that expresses the retrenchment relationship between Ev_A and Ev_C is:

$$R(v, w) \wedge grd_{Ev}(v, w) \wedge BApred_{Ev_C}(w, w') \Rightarrow \exists v' \bullet BApred_{Ev_A}(v, v') \wedge \\ ((R(v', w') \wedge O_{Ev}(v, v', w, w')) \vee C_{Ev}(v, v', w, w'))$$

In this, notice that the grd_{Ev}, O_{Ev} and C_{Ev} clauses have no A/C subscripts; they specify arbitrary joint properties. So the PO says that either O_{Ev} is established, strengthening the joint invariant R, or the concedes relation C_{Ev} is established, useful for exceptional cases. For this to be useful, it is often the case that R is trivialised to true, with the grd_{Ev} and O_{Ev} relations expressing any needed non-trivial relationship between the state spaces, as needed by various retrenchment-related event pairs.

3 A Simple Switching Example

Our case study considers a simplified switching application. We model the switching in or out of some high consequence apparatus. This might concern high voltage equip-

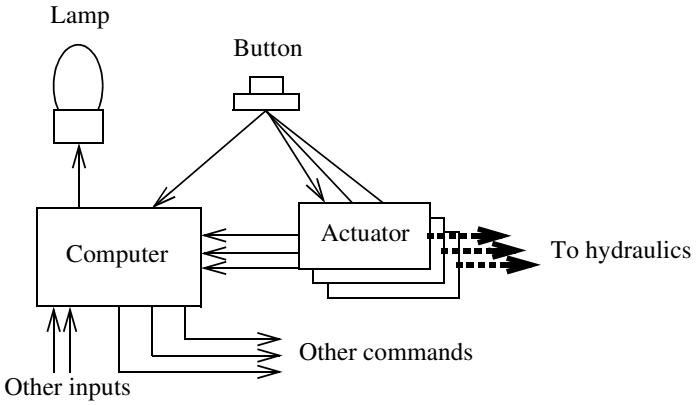

Fig. 1. A simplified switching mechanism.

ment, or bulk gas transportation, or heavy duty water management switching machinery, etc. Indeed, any situation in which the energetics of the physical elements of the system requires management of large quantities of energy, whereas the switching commands —computer mediated as is invariably the case these days— are energetically negligible by comparison, is a potential application. Fig. 1 shows our system. There is a button, pressed by the operator, to switch the apparatus on (and to switch it off again).

The button command is sent to the computer that controls the functioning of the various electro-mechanical components of the switching apparatus. The computer does this via a number of sensor inputs and actuator outputs, most of which we ignore here. Once the operations of the switching have completed, the computer sends a signal telling the lamp to light, confirming success to the operator.

The button command is also sent to a triplicated hydraulic actuator to power up the hydraulics that will cause the movement of the electromechanical components needed for the actual switching of the apparatus. Successful powerup of the hydraulics is signalled to the computer from a sensor in each hydraulic actuator. This signal acts as a safety interlock that helps prevent malfunction of the system (which could result in costly damage to the equipment). Thus, if the hydraulics fails, the absence of the sensor signal prevents the computer from issuing further commands, avoiding damage. Likewise, if the computer fails, the mere powerup of the hydraulics does not cause anything to move, and again, damage is avoided. Such mutual confirmation is a common feature of high criticality systems. To permit discussion of degraded behaviour, we assume the hydraulic actuators (alone among all the components) are triplicated. This is not very convincing from a critical systems perspective but helps keep our example small.

Fig. 2 contains a top level model of the system, machine $Switch$. There are variables btn and $lamp$ representing the button and lamp. Both are binary. The $ButtonDown$ event is permitted when everything is off, and puts the button *D*ow*N*. Likewise, the $ButtonUp$ event is permitted when everything is on, and puts the button *UP*. With the button down, the lamp can come *ON* whereas if the button is up, the lamp can go *OFF*. The guards on $ButtonDown$ and $ButtonUp$ prevent race conditions when switching on has not completed before switching off is commanded, etc. This is the operator's view of the system.

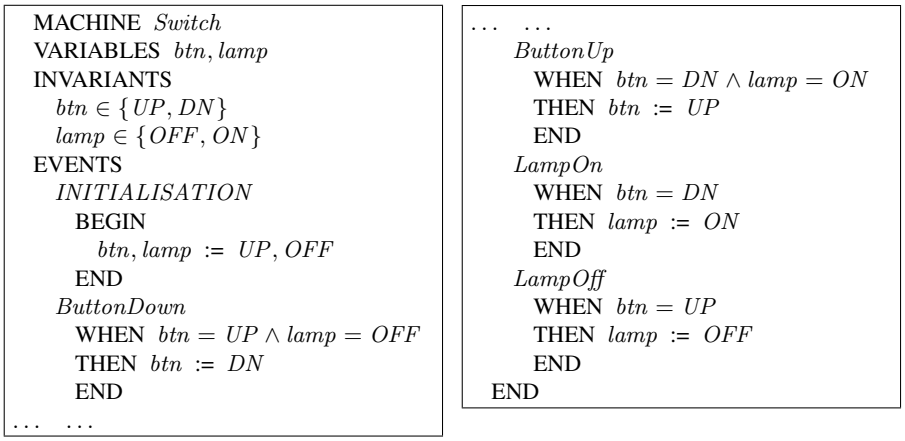

Fig. 2. Top level (operator's) model of the power switching system.

Note the invariants of machine $Switch$. They are just typing invariants. Brief inspection of the events convinces that all four combinations of the two values for the two variables are possible, so there are actually *no* non-trivial invariants at this level with the events as we have defined them.

In Figs. 4 and 5, the $Switch$ machine is refined to $SwitchR$ to include the additional detail described above. Consequently, there are many events to propagate the operator's commands into the system. Fig. 3 shows the intended aspects of this by showing the dependencies among the various variables. Thus, a $ButtonDown$ command to the btn variable enables corresponding commands to the $btnC$ (computer) variable, and to the replicated $btnH_i$ (actuator) variables ($i \in \{1, 2, 3\}$). The latter enable commands to $hydr_i$. Both $btnC$ and $hydr_i$ are needed to enable the $lampsig$ command, which is in turn needed to enable the $lamp$ itself. The sequence for $ButtonUp$ is similar.

We comment on the invariants of $SwitchR$. When a switching on or switching off process is in progress, little can be said about the relationships between the values of variables. A particular value of a given variable in Fig. 3 does not imply anything non-trivial about the value of any other variable. However, if we have a set of values for variables across a *cut* of Fig. 3 consistent with a $ButtonDown$ command, and a set of values for variables across a *later cut* of Fig. 3 also consistent with a $ButtonDown$ command, then we can know that all the variables between the two cuts must also have values consistent with a $ButtonDown$ command, because a $ButtonDown$ command must complete fully, before a $ButtonUp$ command can start (this is clear from the guards on $ButtonDown$ and $ButtonUp$). This allows us to write down a whole collection of invariants.

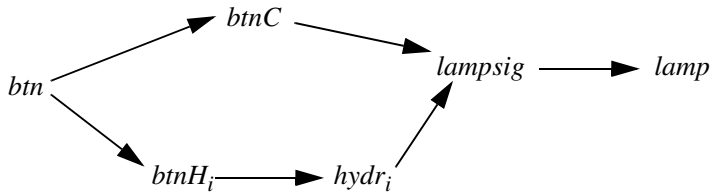

Fig. 3. Variable dependencies in the refined model.

MACHINE $SwitchR$
REFINES $Switch$
VARIABLES
$btn, btnC, btnH_i, hydr_i,$
$lampsig, lamp$
INVARIANTS
$btn \in \{UP, DN\}$
$btnC \in \{UP, DN\}$
$\bigwedge_i btnH_i \in \{UP, DN\}$
$lampsig \in \{OFF, ON\}$
$lamp \in \{OFF, ON\}$
$\bigwedge_i hydr_i \in \{OFF, ON\}$

$btn = DN \wedge lamp = ON \Rightarrow$
$btnC = DN \wedge \bigwedge_i btnH_i = DN \wedge$
$\bigwedge_i hyrd_i = ON \wedge lampsig = ON$
$btn = DN \wedge lampsig = ON \Rightarrow$
$btnC = DN \wedge \bigwedge_i btnH_i = DN \wedge$
$\bigwedge_i hyrd_i = ON$
$btn = DN \wedge btnC = DN \wedge$
$\bigwedge_i hyrd_i = ON \Rightarrow \bigwedge_i btnH_i = DN$
$btnC = DN \wedge \bigwedge_i btnH_i = DN \wedge$
$lamp = ON \Rightarrow$
$\bigwedge_i hyrd_i = ON \wedge lampsig = ON$
$btnC = DN \wedge \bigwedge_i btnH_i = DN \wedge$
$lampsig = ON \Rightarrow \bigwedge_i hyrd_i = ON$
$btnC = DN \wedge \bigwedge_i hyrd_i = ON \wedge$
$lamp = ON \Rightarrow lampsig = ON$

$btn = UP \wedge lamp = OFF \Rightarrow$
$btnC = UP \wedge \bigwedge_i btnH_i = UP \wedge$
$\bigwedge_i hyrd_i = OFF \wedge lampsig = OFF$
$btn = UP \wedge lampsig = OFF \Rightarrow$
$btnC = UP \wedge \bigwedge_i btnH_i = UP \wedge$
$\bigwedge_i hyrd_i = OFF$
$btn = UP \wedge btnC = UP \wedge$
$\bigwedge_i hyrd_i = OFF \Rightarrow \bigwedge_i btnH_i = UP$
$btnC = UP \wedge \bigwedge_i btnH_i = UP \wedge$
$lamp = OFF \Rightarrow$
$\bigwedge_i hyrd_i = OFF \wedge lampsig = OFF$
$btnC = UP \wedge \bigwedge_i btnH_i = UP \wedge$
$lampsig = OFF \Rightarrow \bigwedge_i hyrd_i = OFF$
$btnC = UP \wedge \bigwedge_i hyrd_i = OFF \wedge$
$lamp = OFF \Rightarrow lampsig = OFF$

... ...

... ...
EVENTS
$INITIALISATION$
REFINES $INITIALISATION$
BEGIN
$btn, btnC := UP, UP$
$||_i\ btnH_i, hydr_i := UP, OFF$
$lampsig, lamp := OFF, OFF$
END
$ButtonDown$
REFINES $ButtonDown$
WHEN $btn = UP \wedge lamp = OFF$
THEN $btn := DN$
END
$ButtonDownC$
WHEN $btn = DN$
THEN $btnC := DN$
END
$ButtonDownH_i$
WHEN $btn = DN$
THEN $btnH_i := DN$
END
$ButtonUp$
REFINES $ButtonUp$
WHEN $btn = DN \wedge lamp = ON$
THEN $btn := UP$
END
$ButtonUpC$
WHEN $btn = UP$
THEN $btnC := UP$
END
$ButtonUpH_i$
WHEN $btn = UP$
THEN $btnH_i := UP$
END
$HydraulicsOn_i$
WHEN $btnH_i = DN$
THEN $hydr_i := ON$
END
$HydraulicsOff_i$
WHEN $btnH_i = UP$
THEN $hydr_i := OFF$
END
... ...

Fig. 4. The *SwitchR* machine, a refinement of *Switch* to include additional modelling detail.

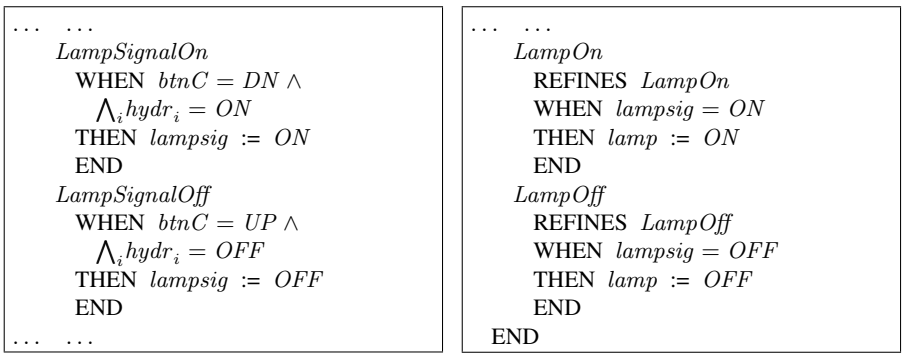

Fig. 5. The *SwitchR* machine, continued.

In the left box of Fig. 4 typing invariants occur above the first horizontal line. Beneath are some invariants based on the above observations. So the first one is based on a cut through btn and another through $lamp$ (defining the hypothesis of the implication) and the conclusion claims values for all the other variables, these all being values consistent with a $ButtonDown$ process. The next invariant moves the second cut to $lampsig$. And so on, until the cuts meet in the middle leaving no variable in between whose value is to be claimed in the conclusion. Note that we have not even exhausted the possibilities for generating invariants according to the scheme described, since all our invariants either include (or exclude) values for all three $btnH_i$ variables together. There will be additional invariants in which some $btnH$ variable(s) is/are hypothesised and the other(s) is/are concluded, and vice versa. Likewise for the $hydr_i$ variables. Between the second and third horizontal lines are analogous invariants for the $ButtonUp$ process, given by inverting all the variable values.

Whether or not a genuine engineering process would choose to assert all these invariants is open to discussion. Provided they are true, their proof lends some additional assurance to, and confidence in, the correctness of the model. But that is not the only issue. The proofs may come at some cost in labour to establish their truth, so the effort that must be invested in doing such verification must be weighed against the additional assurance to be gained, in a cost-benefit analysis.

In any event, overwhelmingly often, the invariants that one chooses to include in a development are precisely that: i.e. a matter of choice and judgment. The choice might be influenced by many things, not the least of these being the ease with which one or other system property is capable of being expressed in the formalism being used for the system model. In the vast majority of cases, the invariants express only a safe approximation to the reachable set of the state space, so *which* safe approximation is chosen to be represented via the invariants is not an absolute and immutable attribute of the problem. (Despite the obvious truth of this, it is surprising how often we speak of *the* invariants, as though there were no choice in the matter.)

4 Switching under Degraded Operation

We now develop our switching application to include some failure modes. We permit failures in the actuators, but these are the only components in our development that we

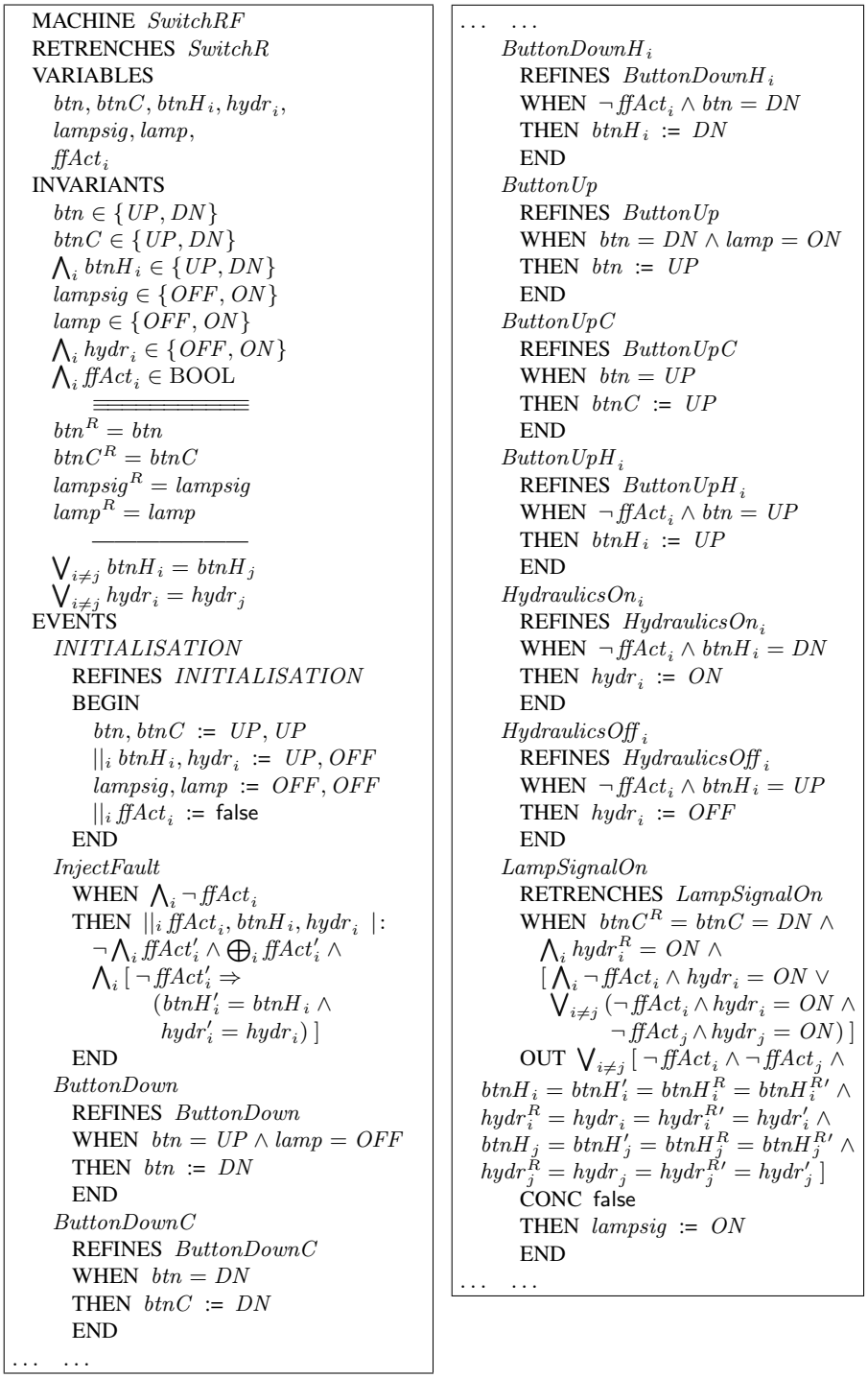

Fig. 6. The *SwitchRF* machine, a retrenchment of *SwitchR* to include failure modes.

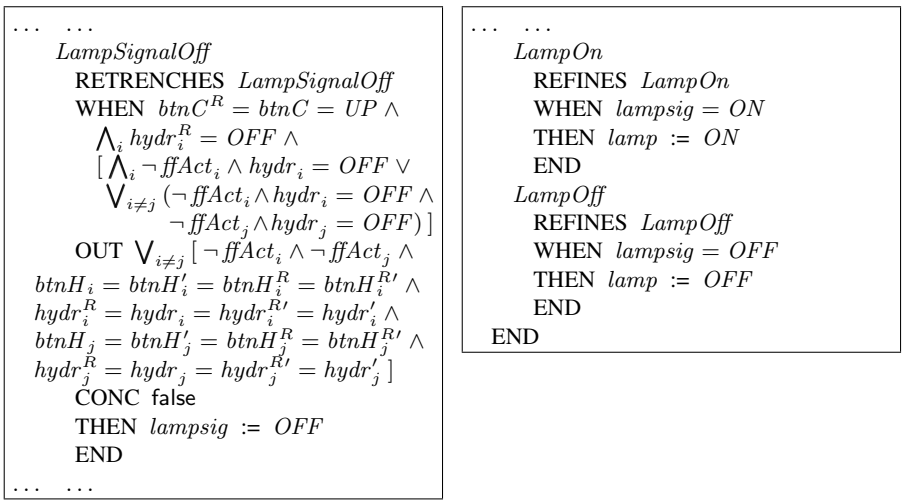

Fig. 7. The *SwitchRF* machine, continued.

permit to fail. Although this is not very convincing in an engineering sense, it helps keep the development to a size we can accommodate in this paper, thus illustrating the brief remarks about our general technique made in the Introduction. Although we introduce and tolerate some faults, we do not go so far as to recover from them, or to model faults beyond the tolerable regime we define.

Figs. 6 and 7 show the faulty machine $SwitchRF$. It retrenches $SwitchR$ since the difference in behaviour compared with $SwitchR$ is too great to reconcile via refinement. There are three new boolean variables $ffAct_i$ to model the presence of a fault in one of the actuators.

The invariants of $SwitchRF$ start with the usual typing invariants. The three horizontal lines suggest the two large blocks of 'two cuts' invariants that we have removed from the $SwitchR$ version of the machine. We discuss these a little later.

Next come the joint invariants. In situations where we deal with behaviours as different as nominal vs. faulty, and use retrenchment, it is very important to maintain the distinction between the two models. The easiest way is to ensure all variable names are disjoint between the two models. So we have renamed the $SwitchR$ variables by attaching an '^{R}' superscript, and have kept the $SwitchRF$ variables undecorated. So $btn^R = btn$ declares that the $SwitchR$ and $SwitchRF$ versions of the button variable always have the same value, etc. Evidently, since the actuators can fail, the $btnH_i$ and $hydr_i$ variables do not figure among the non-trivial joint invariants.

Below the horizontal line come two examples of the kind of weaker invariants that express the maximum that may be sustainable in the presence of faults. They say that for at least one two-out-of-three combination of actuator variables, the $btnH$ variables will agree, and the $hydr$ variables will agree. When both kinds of variable are two-valued, and there are three of each to compare to each other, this cannot help but be true.

Among the events, after initialisation, there is the fault injection event $InjectFault$, which features some unfamiliar syntax. It is only enabled when there have been no faults

hitherto ($\bigwedge_i \neg \mathit{ffAct}_i$), and it specifies its update using Event-B's 'arbitrary assignment satisfying a predicate' operator '$|:$'. Thus, the after-values (primed) of all the ffAct_i, $btnH_i$, $hydr_i$ variables are assigned to satisfy the clauses that follow. Firstly, exactly one of the ffAct'_i variables is set to true via the negated overall and combined with the exclusive or over the three variables (thus disabling $InjectFault$ in future). Secondly, whenever a fault variable ffAct'_i is unset, then the after-values of the corresponding $btnH_i$ and $hydr_i$ variables do not change from their before-values. Saying nothing about the after-values of the $btnH_i$ and $hydr_i$ variables for the ffAct'_i variable that *is* set, allows them to be assigned arbitrarily within their type.

Turning to the other events, the $ButtonDown/Up(C)$ events are unaffected. Moreover, the $ButtonDown/UpH_i$ and $HydraulicsOn/Off_i$ events are enabled only when there is no fault in the corresponding actuator. Since, for these events, the only change is this strengthening of the enabledness condition, these events refine their $SwitchR$ versions according to Event-B rules. The $LampOn/Off$ events are also unchanged.

The fact that the actuator events are disabled by faults, and once a fault has arisen, no further change in the faulty actuator can take place, makes our model a reasonable depiction of a '*stuck_at*' type of fault at our level of abstraction.

This leaves the $LampSignalOn/Off$ events. Here we need to relate the $SwitchR$ behaviours to the $SwitchRF$ behaviours. The guards demand agreement between the $btnC^{(R)}$ clauses, since the computer does not fail. Beyond this, the $SwitchR$ behaviours demand all three $hydr_i^R$ variables fix on the same value before enabling the event, whereas in the $SwitchRF$ behaviours, either there is no fault, and the behaviour is as for $SwitchR$, or there is a fault, and fault tolerance requires that two non-failing actuators agree on a value to enable the event.

All this is recorded in the WHEN clauses.[2] Thus, the WHEN clauses contain the conjunctions of the three $hydr^R$ conditions from $SwitchR$, alongside a disjunction between, either firstly: the conjunction of the corresponding three fault-free $\neg\, \mathit{ffAct} \wedge hydr$ conditions from $SwitchRF$, or secondly: of any two-out-of-three combination of fault-free $SwitchRF$ conditions. The enabled events then set $lampsig$ appropriately.

The OUT clauses of the two events declare some facts about the state of affairs upon event completion. It is claimed that the before- and after- values, in both the $SwitchR$ and $SwitchRF$ machines, of the $hydr$ variables, are all equal, for both the i'th and j'th non-failing actuators. This is evidently true from the definitions of the events (i.e. it follows from the assumed WHEN clause and the $lampsig$ update).

The same is claimed for the $btnH$ variables. This is also true for our system, since we could only arrive at a situation in which the $SwitchRF$ event is enabled, when two non-failing actuators in the $SwitchR$ and $SwitchRF$ systems have followed the same trajectory. The proof would depend on the fact that the only way to set the enabling $btnC$ variables, is via the $HydraulicsOn/Off$ events and these can only execute with the $btnH$ variables at the right values. Actually establishing this mechanically would invariably entail the creation of additional invariants to express this fact, which could then be used in the proof of the OUT properties. (This reinforces what was said about invariants at the end of Section 3.)

[2] The generalised kind of guard used in retrenchment is typically referred to as the WITHIN clause in work on retrenchment.

There is no non-trivial CONCession clause in these events. The overall updates are refining, even if 'non-refining' behaviour is seen in the events themselves — our approach was to merely define behaviour for the $HydraulicsOn/\overline{Off}$ events that could sidestep the faults that are permitted in the system. Non-trivial concessions clauses would most likely be needed if we had defined additional behaviour to react to and recover from detected faults.[3]

It is relatively self-evident that the retrenchment PO quoted at the end of Section 2 is provable for the $HydraulicsOn/\overline{Off}$ events, and that it defaults to the refinement PO for the other, refining events of the $SwitchRF$ machine.

5 Strong Invariants in Degraded Operation

Suppose we now took the $SwitchRF$ machine and removed the $InjectFault$ event. Let us call the resulting machine $SwitchRF_{\overline{IF}}$. In $SwitchRF_{\overline{IF}}$, since no fault can arise, the faulty states catered for in the various events become unreachable. So all the invariants of the nominal $SwitchR$ machine become true again in $SwitchRF_{\overline{IF}}$. So, suppose we now took the $SwitchRF_{\overline{IF}}$ machine and added to it the blocks of strong invariants between the horizontal lines of Fig. 4. Let us call the resulting machine $SwitchRF_{\overline{IF}}INV$. Then $SwitchRF_{\overline{IF}}INV$ is a correct machine. Thus, we have shown that by performing a relatively superficial syntactic transformation on the faulty system model, we have been able to reintroduce strong invariants that held only in the nominal model. We are done!

It is important to note that the procedure just outlined affects only the syntactic periphery of the $SwitchRF$ machine. We took out the $InjectFault$ event, which is isolated from the other events in the sense that it is always enabled (until it disables itself), and does not interact with any of the complex functional interdependencies that the internal parts of a complicated design would display. A lot of work would have gone into the creation of the faulty model in a realistic engineering problem, and we do not wish to interfere with that for the sake of some additional assurance, especially if that would involve the risk of inadvertently introducing some inconsistency with the true model. Aside from the $InjectFault$ removal, we added invariants, which again does not interfere with the internal structure of the true faulty model. Any useful variation of the process we are describing needs to be non-invasive in the manner we have suggested.

Still, before we get too elated about this outcome, we underline that we said that the invariants added to the $SwitchRF_{\overline{IF}}$ machine to get the $SwitchRF_{\overline{IF}}INV$ machine were true, not that they were provable (at least not necessarily provable relatively easily). This is related to the invariants being a safe approximation to the reachable set, in a kind of converse.

Normally, all the events we write in a system model, and all of their capabilities, represent what we desire our system model to actually be capable of. In the case of a faulty model like the $SwitchRF$ machine, we purposely added faulty capabilities to the events because we wished to be able to handle that kind of behaviour in a real

[3] We write 'non-refining' in quotes because we have carefully defined the joint invariants to be oblivious to the faulty behaviour.

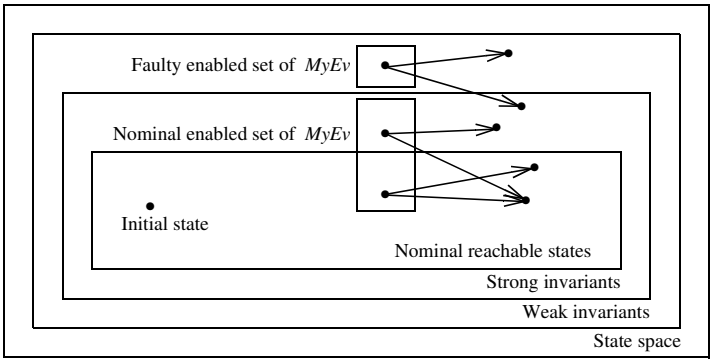

Fig. 8. State space, weak and strong invariants, and enabled sets for a faulty event $MyEv$.

world implementation. But when we disable fault activation, the parts of events' behaviours relevant to the faulty aspects can never be reached, because the relevant part of the events' enabled sets are outside the reachable set of the modified system. Thus, though events are still defined as being capable of faulty behaviour, that behaviour can never arise in a run of the model. Thereby the stronger invariants of the nominal model become true, even in the context of the faulty capability.

Fig. 8 gives a picture of the situation. It shows the total state space, the states described by the weak invariants relevant to faulty behaviour, the states described by the strong invariants relevant to nominal-only behaviour, and the nominal reachable states. Once faults have been deactivated, only states in the nominal reachable set can arise in any run of the system. So for a putative potentially faulty event like $MyEv$, having an enabled set containing faulty as well as nominal states, only the nominal enabled states need to be checked to preserve the invariants, and it is the strong invariants that will hold. That the nominal enabled states can be neatly partitioned from the faulty enabled states follows from the fact that all variables are discrete valued.

The above discussion relies heavily on observations concerning global reachability. This always makes event-by-event reasoning, which is the paradigm in formalisms like Event-B, more difficult. In the context of our toy case study, the remedy is not hard to find. In $SwitchRF_{\overline{IF}}INV$, since the initialisation sets all fault variables to false, and no event changes any fault variable, the falseness of the fault variables becomes an invariant, which is immediately provable, and which may therefore be added to the machine invariants. Since all unruly behaviour by any event involves fault variables that have the value true in a non-trivial manner, unruly behaviour is immediately ruled out during the proof of the simulation PO by the assumed invariants in the PO hypothesis. So only nominal cases need to preserve the invariants, which *should* follow from the properties of the nominal behaviour inside the faulty event. The '*should*' crystalises the additional assurance in the design that our process is intended to provide.

Still, before we get too elated about the preceding conclusion, we should note that it relies on the explicit presence of the fault variables inside the details of the events of the faulty model. The trouble is that the fault variables are really a fiction that abstracts from more detailed model behaviour at lower levels of description. Although a fault in-

jection technique will always require some kind of external event, such as $InjectFault$ to initiate the faulty behaviour, at lower levels of description, the ensuing activity will normally involve the variables that describe how the system is constructed, using sensor data values and similar quantities. Moreover, the system will infer some approximation to the value of the fault variables indirectly, by correlating these sensor and other internal values, to determine the best guess at the actual system state.

In a situation as just described, it may be a lot harder to infer the effective values of fault variables, than when fault variable values are available directly. Thus, a proof that only nominal cases can execute and that they preserve the stronger invariants of the nominal model becomes much harder to carry through. Inevitably, more complicated cases will require reasoning about larger or smaller fragments of system runs, and this makes life much more taxing for an approach that strives to divide and conquer the problem of whole system verification by breaking it down into a *per event* proof activity. In such cases, per-event reasoning entails the creation of a host of additional invariants, which capture the properties of the progress through the required fragment of a system run in a relatively finegrained way. Discovering the needed additional invariants can become very non-trivial. Observations such as these lead us to alternatively advocate approaches based around k-induction [27, 18] as potentially offering reasonable possibilities for outflanking maneuvres, if a direct 1-inductive proof of the desired invariants becomes sufficiently challenging.

With the above caveat on board, we can summarise our approach to the gaining of additional assurance in the design of faulty system models in the following way.

- Develop the nominal model first. Populate it with as many strong invariants as desirable or useful.
- Develop the faulty model via a set of functional departures from the nominal model, and by removal of invariants that only hold in the nominal model. Retrenchment provides a useful vehicle for this.
- Ensure faults are activated in the faulty model in a way that is easy to disable non-invasively, e.g by using fault injection.
- Disable the activation of faults and reintroduce the removed strong invariants.
- Reprove the model.

6 Related Work

Overtly formal techniques, based on confirming that invariants attributed to a system model hold, and fault engineering, based on considering behaviours that violate such invariants, are like oil and water. Overwhelmingly, formal techniques, even when not proving that invariants hold, are concerned with issues involving consistency of invariants and behaviour, such as the process of synthesising invariants from behaviour. In this vein we can mention [14, 20, 19, 23], among many others.

The formal treatment of safety (including fault tolerance) throws up issues similar to ones treated in this paper. See for example [16, 15]. The connection with the retrenchment approach can be seen in [9, 10].

Also close to the approach of this paper is the work on the KAOS requirements methodology [21, 22]. Although focused on requirements, behind the scenes it is highly

formalised, and the kinds of departures from ideal behaviour that are unavoidable when debating a family of requirements and that are comparable to our failure modes are, in KAOS, termed *obstacles*. The parallels between the KAOS approach and what is done in retrenchment are described in [8].

7 Conclusions

In the preceding sections we summarised Event-B, and in particular, its refinement and retrenchment POs. Discharging these forms the crux of the Event-B development method. We then introduced a simple switching example to exemplify the subsequent discussion. It was intended to be small enough that we could accommodate it within this paper, yet was complex enough that it admitted the kind of non-trivial invariants that we wanted to consider.

We started with a controller's view and refined it to a more detailed internal model, though still only exhibiting nominal behaviour. We argued for the validity of a large number of strong invariants in this nominal model. We then introduced faults into the model via fault injection, using the retrenchment technique. The latter offers some POs to help control the process of relatively arbitrary system model change. With faults included, we had to remove the strong invariants, which would fail any attempt to verify them. This gave us the springboard for discussing the controlled reintroduction of the strong invariants when faults were deactivated. Provided the deactivation can be done in a suitably non-invasive manner, the resulting model ought to be correct.

We pointed out that establishing the correctness of the resulting model may be easy if the model involves fault variables explicitly, but may be more difficult if all behaviour is exclusively expressed in terms of real system quantities. Thus the tractability of our proposed technique can vary greatly depending on the level of abstraction at which it is applied. That said, in all developments it should be possible to identify a level of abstraction which is high enough so that a system model pitched at that level will prove tractable as regards the applicability of the technique, and thus, to apply the technique there. Refinement may then be sufficient to propagate the guarantess obtained to lower levels of abstraction, if needed.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. Int. J. Soft. Tools for Tech. Trans. 12(6), 447–466 (2010)
3. ADVANCE: European Project ADVANCE. IST-287563 http://http://www.advance-ict.eu/
4. Back, R., Sere, K.: Stepwise Refinement of Action Systems. In: Proc. MPC-89. vol. 376, pp. 115–138. Springer, LNCS (1989)
5. Back, R., von Wright, J.: Trace Refinement of Action Systems. In: Proc. CONCUR-94. vol. 836, pp. 367–384. Springer, LNCS (1994)
6. Back, R., Kurki-Suonio, R.: Decentralisation of process nets with centralised control. In: 2nd ACM SIGACT-SIGOPS Symposium on PODC. pp. 131–142. ACM (1983)

7. Back, R., Sere, K.: Superposition Refinement of Reactive Systems. Formal Aspects of Computing 8(3), 324–346 (1996)
8. Banach, R.: A Deidealisation Semantics for KAOS. In: Lencastre, M. (ed.) Proc. ACM SAC-10 (RE track). pp. 267–274. ACM (2010)
9. Banach, R., Bozzano, M.: The Mechanical Generation of Fault Trees for Reactive Systems via Retrenchment I: Combinational Circuits. Form. Asp. Comp. 25, 573–607 (2013)
10. Banach, R., Bozzano, M.: The Mechanical Generation of Fault Trees for Reactive Systems via Retrenchment II: Clocked and Feedback Circuits. Form. Asp. Comp. 25, 609–657 (2013)
11. Banach, R., Jeske, C.: Retrenchment and Refinement Interworking: the Tower Theorems. Math. Struc. Comp. Sci. 25, 135–202 (2015)
12. Banach, R., Jeske, C., Poppleton, M.: Composition Mechanisms for Retrenchment. J. Log. Alg. Prog. 75, 209–229 (2008)
13. Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Engineering and Theoretical Underpinnings of Retrenchment. Sci. Comp. Prog. 67, 301–329 (2007)
14. Beyer, D., Henzinger, T., Majumdar, R., Rybalchenko, A.: Invariant Synthesis for Combined Theories. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 378–394. Springer (2007)
15. Bittner, B., Bozzano, M., Cavada, R., Cimatti, A., Gario, M., Griggio, A., Mattarei, C., Micheli, A., Zampedri, G.: The xSAP Safety Analysis Platform. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 533–539. Springer (2016)
16. Bozzano, M., Villafiorita, A.: Design and Safety Assessment of Critical Systems. CRC Press (Taylor and Francis), an Auerbach Book (2010)
17. DEPLOY: European Project DEPLOY. IST-511599 http://www.deploy-project.eu/
18. Donaldson, A., Haller, L., Kroening, D., Rümmer, P.: Software Verification Using k-Induction. In: Proc. SAS-11. vol. 6887, pp. 351–368. Springer, LNCS (2011)
19. Furia, C., Meyer, B.: Inferring Loop Invariants Using Postconditions. In: Fields of Logic and Computation, pp. 277–300. Springer (2010)
20. Ghilardi, S., Ranise, S.: Goal-Directed Invariant Synthesis for Model Checking Modulo Theories. In: International Conference on Automated Reasoning with Analytic Tableaux and Related Methods. pp. 173–188. Springer (2009)
21. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley (2009)
22. Letier, E.: Reasoning about Agents in Goal-Oriented Requirements Engineering. Ph.D. thesis, Dépt. Ingénierie Informatique, Université Catholique de Louvain (2001)
23. Qin, S., He, G., Luo, C., Chin, W.N.: Loop Invariant Synthesis in a Combined Domain. In: International Conference on Formal Engineering Methods. pp. 468–484. Springer (2010)
24. RODIN: European Project RODIN (Rigorous Open Development for Complex Systems) IST-511599 http://rodin.cs.ncl.ac.uk/
25. Sekerinski, E., Sere, K.: Program Development by Refinement: Case Studies Using the B-Method. Springer (1998)
26. Voisin, L., Abrial, J.R.: The Rodin Platform has Turned Ten. In: Ait Ameur, Schewe (eds.) Proc. ABZ-14. vol. 8847, pp. 1–8. Springer, LNCS (2014)
27. Wahl, T.: The k-induction principle. http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf