# Core Hybrid Event-B II: Multiple Cooperating Hybrid Event-B Machines

Richard Banach[a,1], Michael Butler[b], Shengchao Qin[c], Huibiao Zhu[d,2]

[a]*School of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, U.K.*
[b]*School of Electronics and Computer Science, University of Southampton, Highfield, Southampton, SO17 1BJ, U.K.*
[c]*School of Computing, University of Teesside, Middlesbrough, Tees Valley, TS1 3BA, U.K.*
[d]*Shanghai Key Laboratory of Trustworthy Computing, MOE International Joint Laboratory of Trustworthy Software, International Research Center of Trustworthy Software, East China Normal University, Shanghai 200062, China.*

## Abstract

Hybrid Event-B, initially introduced for single machines to add continuously varying behaviour to discrete change of state in Event-B, is extended to cater for multiple cooperating machines. Multiple machine working is mediated by INTERFACE and PROJECT constructs. The former encapsulates a set of variables, their invariants and initialisations, in a form that several machines can exploit simultaneously. The latter organises the set of cooperating machines and interfaces into a coherent system. Machine instantiation and composition via interfaces are discussed. Machine decomposition is explored in this framework. Multi-machine refinement is described. A hypergraph project architecture is proposed. Two small case studies, on power switching and on the European Train Control System (the latter treated earlier within the single machine formalism), illustrate these mechanisms. The semantics of interacting multi-machine systems is described, and proof obligations that ensure correctness are covered.

## 1. Introduction

In [9], henceforth referred to as PaperI, we introduced Hybrid Event-B for single machines. This enhancement of discrete Event-B was intended to address the unavoidable involvement of continuously varying behaviour in many of today's hybrid and cyber-physical systems. In that paper, we explored the background and motivations for doing the design in the way we did it, and we gave both an informal account of how Hybrid Event-B worked, and a formal semantics, along with a discussion of refinement.

Of course, single machines are not enough. The cyber-physical systems we mentioned, are, these days, highly interconnected interacting multi-agent systems, coupling together a typically very heterogeneous collection of subsystems. A Hybrid Event-B design that caters for such situations should be able to model separate subsystems as separate machines, independent and yet interconnected in ways that look convincing from the application domain perspective, and with the ability to impose appropriate invariants on the system as a whole, including invariants that suitably couple together mutually dependent machines.

*Email addresses:* `banach@cs.man.ac.uk` (Richard Banach), `mjb@ecs.soton.ac.uk` (Michael Butler), `s.qin@tees.ac.uk` (Shengchao Qin), `hbzhu@sei.ecnu.edu.cn` (Huibiao Zhu)

[1]A portion of the work reported in this paper was done while the first author was a visiting researcher at the Software Engineering Institute at East China Normal University. The support of ECNU is gratefully acknowledged.

[2]Huibiao Zhu is supported by National Natural Science Foundation of China (No. 61361136002) and Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (ZF1213).

In this paper we present such an extension. There are two main questions that it has to address. The first is structural/syntactic: how is the information that describes a large system divided up among its components, and how do these relate to each other? The second is semantic: how is the semantics of a single machine altered by the presence of other machines, and what is the semantics of interaction?

Briefly, we answer the first question by introducing new syntactic constructs, the INTERFACE and the PROJECT. The former encapsulates a set of variables, their invariants and their initialisations; this allows multiple machines to access these ingredients in a disciplined way. The latter defines which machines and interfaces contribute to a project, and takes care of instantiation issues. We answer the second question by noting that the mathematics that underlies the semantics of a single machine can be extended fairly readily to several machines, the main additional complication coming from choosing enabled events to execute *per machine* rather than on a global basis. Moreover, we introduce synchronised events, which are collections of events in different machines which must execute together, the semantics of this being unproblematic due to the *in principle* applicability of the single machine semantics to projects as a whole. The actual specification of which combinations of events need to synchronise is delegated to the PROJECT construct.

The rest of this paper is as follows. Section 2 gives an overview of single machine Hybrid Event-B, for orientation. Section 3 gives a brief overview of PROJECTs, INTERFACEs and MACHINEs. Section 4 explores interfaces in detail, describing how a community of machines may share variables, and the combinatoririal rules that govern this. Section 5 covers refinement in this context. Section 6 discusses issues of synchronisation for Hybrid Event-B in general terms. Next, Section 7 covers projects in detail, giving syntactic precision to the various technical mechanisms described thus far. Section 8 then covers how a machine may be decomposed once successive refinements have made it inconveniently big. This includes partitioning of the components into submachines, and the decomposition of individual events, as needed. Section 9 describes the hypergraph project architecture, giving concrete recommendations on how the mechanisms introduced thus far should be used to best effect. Section 10 illustrates the mechanisms discussed thus far in two small multi-machine case studies. The first covers a simple power switching application; the second is based on the European Train Control System, which was treated as a single machine application in PaperI. Section 11 gives the formal semantics of multi-machine systems. Section 12 discusses correctness in the context of the *per machine* proof obligations (POs) treated in PaperI. Section 13 summarises the detailed changes needed to these single machine POs to enable them to work in the multi-machine context. Section 14 concludes.

## 2. Single Machine Hybrid Event-B, a Sketch

In this section, for purposes of orientation, we briefly review the single machine Hybrid Event-B formalism of PaperI [9], inevitably glossing over many points covered more carefully there.

In Fig. 1 we see a basic Hybrid Event-B machine, *HyEvBMch*. It starts with declarations of time and of a clock. In Hybrid Event-B, time is a first class citizen in that all variables are functions of time, whether explicitly or implicitly. Time is read-only. Clocks allow more flexibility, since they increase like time, but may be set during mode events. Variables are of two kinds. There are mode variables (like $u$) which typically take their values in discrete sets and change their values discontinuously during mode events. There are also pliant variables (such as $x, y$), declared in the PLIANT clause, which typically take their values in topologically dense sets (normally $\mathbb{R}$) and which may change continuously, such change being specified via pliant events.

Next come the invariants. The types of the variables are asserted to be the sets from which the variables' values *at any given moment of time* are drawn. More complex invariants are similarly predicates that are required to hold *at all moments of time* during a run.

```
MACHINE $HyEvBMch$
TIME $t$
CLOCK $clk$
PLIANT $x,y$
VARIABLES $u$
INVARIANTS
  $x,y,u \in \mathbb{R},\mathbb{R},\mathbb{N}$
EVENTS
  $INITIALISATION$
    STATUS ordinary
    WHEN
      $t = 0$
    THEN
      $clk,x,y,u \;:=\; 1,x_0,y_0,u_0$
    END
$\dots\ \dots$
```

```
$\dots\ \dots$
    $MoEv$
      STATUS ordinary
      ANY $i?,l,o!$
      WHERE
        $grd(x,y,u,i?,l,t,clk)$
      THEN
        $x,y,u,clk,o! :|$
          $BApred(x,y,u,i?,l,o!,$
          $t,clk,x',y',u',clk')$
      END
$\dots\ \dots$
```

```
$\dots\ \dots$
    $PliEv$
      STATUS pliant
      INIT $iv(x,y,t,clk)$
      WHERE $grd(u)$
      ANY $i?,l,o!$
      COMPLY
        $BDApred(x,y,u,$
          $i?,l,o!,t,clk)$
      SOLVE
        $\mathcal{D}\,x =$
          $\phi(x,y,u,i?,l,o!,t,clk)$
        $y,o! \;:=$
          $E(x,u,i?,l,t,clk)$
      END
  END
```

Figure 1: A schematic Hybrid Event-B machine.

Then, the events. The *INITIALISATION* has a guard that synchronises time with the start of any run, while all other variables are assigned their initial values as usual.

Mode events are direct analogues of events in discrete Event-B. They can assign all machine variables (except time itself). In the schematic *MoEv* of Fig. 1, we see three parameters $i?,l,o!$, (an input, a local parameter, and an output respectively), and a guard *grd* which can depend on all the machine variables. We also see the generic after-value assignment specified by the before-after predicate *BApred*, which can specify how the after-values of all variables (except time, inputs and locals) are to be determined.

Pliant events constitute the most obvious distinction between Hybrid Event-B and discrete Event-B. They specify the continuous evolution of the pliant variables over an interval of time. The schematic pliant event *PliEv* of Fig. 1 shows the structure. There are two guards: there is *iv*, for specifying enabling conditions on the pliant variables, enabling conditions that mix mode and pliant variables, and clocks and time; and there is *grd*, for specifying enabling conditions purely involving the mode variables. The separation between the two is motivated by considerations connected with refinement.

The body of a pliant event contains three parameters $i?,l,o!$, (again an input, a local parameter, and an output) which are functions of time, defined over the duration of the pliant event. The behaviour of the event is defined by the COMPLY and SOLVE clauses. The SOLVE clause specifies behaviour fairly directly. For example the behaviour of pliant variable $y$ and output $o!$ is given by a direct assignment to the (time dependent) value of the expression $E(\dots)$. Alternatively, the behaviour of pliant variable $x$ is given by the solution of the first order ordinary differential equation (ODE) $\mathcal{D}x = \phi(\dots)$, where $\mathcal{D}$ indicates differentiation with respect to time. The COMPLY clause can be used to express any additional constraints that are required to hold during the pliant event via its before-during-and-after predicate *BDApred*. Typically, constraints on the permitted range of values for the pliant variables, and similar restrictions, can be placed here. Of all time dependent behaviours that satisfy these clauses, only piecewise absolutely continuous functions of time which have both left and right limits at all times, and which are right continuous at all times are considered.

The COMPLY clause has another purpose. When specifying at an abstract level, we do not necessarily want to be concerned with all the details of the dynamics — it is often sufficient to require some constraints to hold which express the needed safety properties of the system during the pliant event. The COMPLY clauses of the pliant events can house such constraints directly, leaving it to lower level refinements to add the necessary details of the dynamics. All the ways of specifying update in either kind of event are referred to as its actions.

Briefly, the semantics of a Hybrid Event-B machine consists of a set of *system traces*, each being a set of functions of time, expressing the value of each machine variable over the duration of a run.

Time is modelled as an interval $\mathcal{T}$ of the reals. A run starts at some initial moment of time, $t_0$ say, and lasts either for a finite time, or indefinitely. The duration of the run $\mathcal{T}$, breaks up into a succession of left-closed right-open subintervals: $\mathcal{T} = \langle [t_0 \dots t_1), [t_1 \dots t_2), [t_2 \dots t_3), \dots \rangle$. The idea is that mode events (with their discontinuous updates) take place at the isolated times corresponding to the common endpoints of these subintervals $t_i$, and in between, the mode variables are constant and the pliant events stipulate piecewise absolutely continuous change in the pliant variables.

Insisting on piecewise absolute continuity is equivalent to demanding that the behaviour over time of each variable is a succession of solutions to well posed initial value problems, e.g. $\mathcal{D}xs = \phi(xs \dots)$, where $xs$ is a relevant tuple of pliant variables and $\mathcal{D}$ is the time derivative [22, 23, 18, 16]. ('Well posed' means that $\phi(xs \dots)$ has Lipschitz constants which are uniformly bounded over a period of absolutely continuous behaviour, bounding its variation with respect to $xs$, and that $\phi(xs \dots)$ is measurable in $t$.) In between the absolutely continuous pieces, isolated discontinuities are acceptable, modelling the actions of mode events, and of discontinuities coming from the environment.

Pursuing this view, a mode transition becomes a before-/after- pair of variable valuations (the change taking place at a single instant), while a pliant transition becomes a time-parameterised set of pairs of such variable valuations (the common before-valuation being at the beginning of the piecewise absolutely continuous behaviour, and the various after-valuations being indexed by the time interval, the changes no longer taking place at a single instant). From this perspective, many questions regarding mode and pliant events and transitions have very similar answers.

Within any interval in which the variables' behaviour is specified by a pliant event, we seek the earliest time at which a mode event becomes enabled, and this time becomes the preemption point beyond which the pliant behaviour is abandoned, and the next pliant behaviour is scheduled after the completion of the mode event.

In this manner, assuming that the *INITIALISATION* event has achieved a suitable initial assignment to variables, a system run is *well formed*, and thus belongs to the semantics of the machine, provided that at runtime:

**[A]** Every enabled mode event is feasible, i.e. has an after-state, and on its completion enables a pliant event (but does not enable any mode event).

**[B]** Every enabled pliant event is feasible, i.e. has a time-indexed family of after-states, and EITHER:

(i) During the run of the pliant event a mode event becomes enabled. It preempts the pliant event, defining its end. ORELSE

(ii) During the run of the pliant event it becomes infeasible: finite termination. ORELSE

(iii) The pliant event continues indefinitely: nontermination.

Thus in a well formed run mode events alternate with pliant events. The last event (if there is one) is a pliant event whose duration may be finite or infinite. The relatively simple view of Hybrid Event-B just expounded, persists in suitable form, in a multi-machine system. We examine the details in Section 11.

## 3. PROJECTs, INTERFACEs, MACHINEs

In this paper, multi-machine systems, henceforth called projects, need various syntactic constructs to specify them. In our scheme there are three kinds of syntactic construct. There are MACHINEs. These are just like the machines described above except that they can access further variables and invariants via CONNECTS or READS clauses. These connect a machine to an INTERFACE construct, which is a container for such shared variables and invariants. There must be a further syntactic construct that collects together all the machine and interface constructs of a project. This is the PROJECT construct.

As well as simply naming all the constituent machines and interfaces, there are various other global coordination issues that are covered in a project file, which we cover later. We will start by discussing the relationship between machines and interfaces.

### *3.1. Name Space Issues*

When we have formal texts such as machines which (at least informally) act as enclosing binders for the free names that occur inside them, and then contemplate combining them into larger entities such as projects, the conventions regarding scopes of names must be clearly understood. In particular, when placing machines inside the bigger project construct, the scopes of various identifiers may need to change so that components are able to interact. The following covers the conventions used in this paper.

Event names remain bound within their machine, and their parameters remain bound with their events (so an input parameter may need to be referred to as $M.Ev.i?$ where $M$ is the machine that contains event $Ev$ of which $i?$ is the input) — however, see Section 7.2 regarding the scopes of such bound parameters in the presence of synchronisations.[3] Variable scopes are extended to include the whole project (so variable name clashes among machines and interfaces are forbidden). Synchronisation identifiers will also be project-wide. This relatively simple scheme allows for uncluttered modelling, and is sufficient for the semantics of multi-machine projects in Section 11. Of course, this does not preclude organising multi-component systems in more sophisticated ways, but these lie beyond the scope of this paper.

### *3.2. Multiproject Situations and Name Spaces*

The provisions of Section 3.1 apply to a single project. However, when we refine a project (or decompose it), we need to specify how names in more than one project relate to each other. We extend the convention just introduced to include the project identifier when many projects are in scope. Thus the earlier $M.Ev.i?$ becomes $Prj.M.Ev.i?$, referring to input $i?$ of event $Ev$ of machine $M$ in project $Prj$. Of course these formal names are not used in concrete syntax. So when variable $x$ occurs in machine $M$ in project $Prj$, and $x$ also occurs in machine $MR$ (that refines $M$) in project $PrjR$ (that refines $Prj$), the two occurrences refer to the formal variables $Prj.M.x$ and $PrjR.MR.x$ respectively, and it is understood that there is a formal (and unstated) joint invariant $Prj.M.x = PrjR.MR.x$ in the refining machine $MR$. (Of course, in practice, all of this is finessed via the one point rule.) Similar remarks apply for interfaces. Also, in common with normal Event-B discourse, unqualified use of the term 'invariant' is intended to always include both machine invariants and joint invariants when both are in scope.

## 4. Hybrid Event-B INTERFACEs

We turn to INTERFACEs. First though, some salutory remarks concerning what is to come.

### *4.1. Complex Mulicomponent Systems*

If we consider a typical multi-component system formalised in a model based way, unless it consists of totally independent components, in general, there will be an infinite number of true facts about it. In general therefore, there will be an infinite number of true invariants that could be written down. The point of saying this is that it is in principle impossible to have written down 'all' the invariants of a nontrivial model. Thus the set of invariants that *is* written down in any particular case, is invariably an outcome of *human judgement* regarding which of the possible invariants deserve to be explicitly noted. We say this because, shortly, we will introduce a pattern for invariants that cut across subsystem boundaries. It will have a fixed generic structure to permit mechanisation. In doing so we accept that not all conceivable invariants will be expressible thereby, so that the utility of our proposal must be borne out by its effectiveness in practice.

[3]Synchronisations are introduced later.

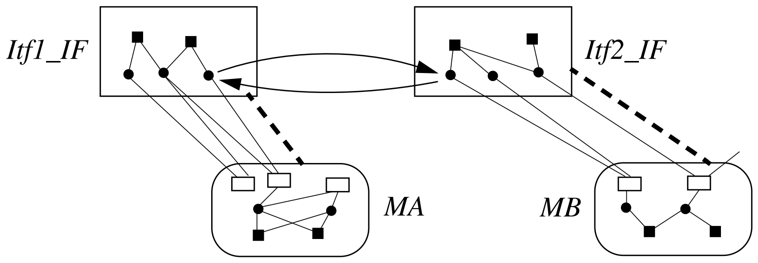

Figure 2: An illustration of the constraints regarding the use of the INTERFACE construct. Machines and interfaces are large rounded and unrounded rectangles respectively. Small rectangles are events; small black circles are variables; small black squares are tIi's. Machines and interfaces contain their variables, tIi's (and events). Arrows connect the local and remote interfaces of a tIIi.

## *4.2. The INTERFACE Construct*

We now introduce INTERFACEs. The word has many connotations, including the idea of interface automata and its close relatives (see e.g. [4]). Our version is rooted in the work of Hallerstade and Hoang in [14], which we extend (in the direction of [4], arguably), to achieve what we need.

An INTERFACE is a syntactic construct that declares some **variables**, and (going beyond [14]), some **invariants** that interrelate them, and their **initialisations**. Any machine that needs to access any of these variables must have a CONNECTS (or READS) clause to it, this being the only way for more than one machine to have access to the same set of variables (and to the related invariants and initialisations).

Consider a Hybrid Event-B project as described in Section 3. Let $\mathbf{V}$ be the set of variables, $\mathbf{I}$ be the set of invariants, and $\mathbf{E}$ be the set of events declared in the project. The rules governing legitimate accessibility/visibility between these various elements are combinatorial in nature, and are described in the following collection of 'diamond rules'.

Let the set of variables $\mathbf{V}$ be partitioned into subsets $\mathbf{V} = A \uplus B \uplus C \uplus \ldots$, such that for every invariant $Inv \in \mathbf{I}$:

**[♦1]** **either** all variables mentioned in *Inv* belong to some subset, eg. $A$;

**[♦2]** **or** the invariant *Inv* is of the form $U(u) \Rightarrow V(v)$, where there are distinct subsets of the partition, $A$ and $B$ say, such that $u$ and $v$ refer to variables in $A$ and $B$ respectively.

We call these type I and type II invariants respectively (tIi and tIIi for short). While tIi's exemplify 'normal invariants', i.e. the vast majority, the tIIi's we have just introduced capture the pattern we spoke of above for the cross-cutting invariants that straddle subsystem boundaries. Aside from global invariants which we introduce later, the tIIi's will be the *only* permitted way to write invariants that are not completely encapsulated by a machine or interface construct. For a tIIi, the associated $A$ and $B$ subsets of $\mathbf{V}$ are known as the local and remote subsets respectively, containing the local variables $u$ and remote variables $v$ respectively. Reinforcing our statement above, we propose that tIIi's are adequate to capture a sufficiently rich class of inter-subsystem properties for practical use.

The lexical nature of the 'diamond' rules in this and related sections means that the 'hybrid' aspect of Hybrid Event-B is not essential for them, see e.g. [6].

Before formally listing the remaining diamond rules, we give a small illustration of the principles in Fig. 2. Small black circles represent variables, while small black squares represent tIi's. Small rectangles represent events. Events and invariants are connected to the variables they involve by thin lines. Interfaces are large rectangles containing the variables and invariants they encapsulate — there are two in Fig. 2, *Itf1_IF* and *Itf2_IF*. Machines are large rounded rectangles, containing their events and local variables — again there are two, *MA* and *MB*. The very short thin line with one end free from an event in *MB* is an I/O variable connected to the environment. The CONNECTS relationship between a machine and an interface is depicted by a thick dashed line. Finally, tIIi's are represented by an arrow from a variable in the local interface to a variable in the remote interface (these containing the local and remote variable subsets of the tIIi respectively). We return to the formal conditions of the interface scheme.

Referring to the partition of **V**, every subset of variables in the partition consists of variables that:

**[♦3]** **either** are declared as the variables of a single machine;

**[♦4]** **or** are declared as the variables of a single interface.

Each interface:

**[♦5]** must contain all the type I invariants that mention any of its variables;

**[♦6]** must contain any type II invariant for which the interface's variables are in the local subset; in each such case the interface must contain a READS *ReadInt* declaration for the (different) interface *ReadInt* that contains the remote variables.

**[♦7]** must contain a REFERS *RefInt* declaration, whenever any of its variables are the remote variables of a type II invariant declared in a (different) interface *RefInt*.

Each machine:

**[♦8]** may declare the variables belonging to a subset of the partition as local (i.e. unshared) variables;

**[♦9]** must contain a CONNECTS *IntRW* declaration whenever any of its events needs (read-and-)write access to the variables of the interface *IntRW*;

**[♦10]** must contain a READS *IntRO* declaration whenever any of its events needs read-only access to the variables of the interface *IntRO*;

**[♦11]** must contain all the type I invariants that mention any of its local variables;

Each event:

**[♦12]** may read and update variables that are declared locally in the machine containing the event, or that are introduced via CONNECTS *IntRW* declarations in the machine containing the event;

**[♦13]** may read (in its guards or in the expressions that define update values) variables that are introduced: either via READS *IntRO* declarations in the machine containing the event, or via READS *ReadInt* or REFERS *RefInt* declarations contained in an interface *IntRW* that the machine containing the event CONNECTS (to).

**[♦14]** must preserve all invariants that are declared in the machine that contains it, or that appear in interface *IntRW* for any CONNECTS *IntRW* declarations of the machine, or that appear in interface *ReadInt* or *RefInt* for any READS *ReadInt* or REFERS *RefInt* declarations contained in any interface *IntRW* that the machine containing the event CONNECTS (to).

Each invariant:

**[♦15]** must be contained in the interface or machine which declares all its variables (if it is a type I invariant), or must be contained in the interface which declares its local variables (if it is a type II invariant).

We can now see that in a Hybrid Event-B project, verifying that all the invariants are preserved by all event executions (provided the initial state satisfies them all), can be easily accomplished using verification conditions that depend on information that is easily located from the syntactic information in the machine that contains the event. We examine verification conditions in Section 12.

Referring to Fig. 2, in addition to the CONNECTS relationship depicted by the thick dashed lines, the READS and REFERS relationships point forwards and backwards respectively along the tIIi arrows.

## 5. Machine and Interface Refinement

Given a project consisting of machines and interfaces satisfying **[♦1]**–**[♦15]**, in this section we consider how its components may be refined. Refinement for Hybrid Event-B machines will be as described for single machines in PaperI. In other words, a refining machine is related to its abstraction using joint invariants relating the variables of both machines (and a host of POs must be verified). Regarding refinement of interfaces, speaking intuitively, we can regard an interface as almost a kind of bare bones machine, lacking explicit events, but implicitly admitting any state update that respects the invariants. In that sense, in interface refinement, variables are refined in the usual way, and joint invariants can be introduced as for a (Hybrid) Event-B machine refinement. Concrete syntax for refinement in both cases can be taken from the Hybrid Event-B scheme (see Section 7). Furthermore, in both cases, the refinement process itself must conform to the following principles.

**[♦16]** The variables of an interface *Itf* must be refined to the variables of its refining interface *ItfR* via a retrieve relation (joint invariant) that mentions only the variables of *Itf* and *ItfR*.

**[♦17]** The variables of a machine *M* must be refined to the variables of its refining machine *MR* via a retrieve relation (joint invariant) that mentions only the variables of *M* and *MR*.

The (essentially) independent refinement of machines and interfaced enforced by **[♦16]**–**[♦17]** prevents the inadvertent falsifying of invariants in situations such as the following counterexample schema.

Suppose each of machines $M_1$ and $M_2$ CONNECTS (to) interface *Itf*. Suppose $M_1$, $M_2$ and *Itf* are refined to $M_1R$, $M_2R$ and *ItfR* respectively. Suppose the joint invariant of the $M_2$ to $M_2R$ refinement involves the variables of *Itf* and *ItfR* too. Then when concrete machine $M_1R$ executes an event, faithful to some abstract event of $M_1$, there is no guarantee that the new state in $M_1$ and $M_1R$ and *Itf* and *ItfR* still satisfies the joint invariants of $M_2$ and $M_2R$ via the coupled joint invariants linking the state in $M_2$ and $M_2R$ to the state in *Itf* and *ItfR*.

However, **[♦16]**–**[♦17]** ensure that any invariant of the $M_2$, $M_2R$, *Itf*, *ItfR* variables is a conjunction of an invariant of $M_2$, $M_2R$ with an invariant of *Itf*, *ItfR*, thus decoupling them, and ensuring that any update to the variables of $M_2$, $M_2R$, or of *Itf*, *ItfR*, that preserves its respective conjunct, preserves both.

Fig. 3 shows a refinement example. Machine *MA* is refined to a larger machine *MM*, containing more variables, invariants and events. The (refinements of) the original elements can be discerned in the figure. Full syntactic details are postponed to Section 7.

## 6. Communication and Synchronisation for Hybrid Event-B Machines

In this section we consider, from a semantic perspective, how the different machines in a project can be efficiently coordinated. Syntactic details are postponed to Section 7. We start by observing that in the field of discrete event formalisms, there are two main ways of coordinating different automata or machines: the shared variable paradigm and the shared (or synchronised) event paradigm.

Focusing on Event-B, the shared variable paradigm is exemplified by some of the original Event-B documentation, and has the longer history. The original idea was described in [1]. However, pursuing this

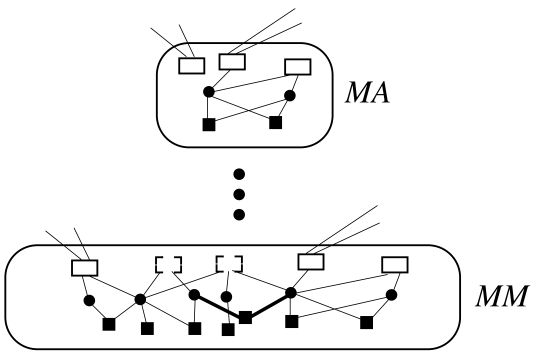

Figure 3: An illustration of the refinement mechanism. *MA* is refined to a larger machine *MM*. (Refinements of) the original variables, invariants and events of *MA* (aside from the leftmost *MA* event) appear on the right in *MM*.

through refinement led to practical difficulties, so refinement of shared variables was forbidden in [15]; but this impedes flexible development. A more streamlined approach to the problem was [14], which introduced interfaces similar to ours, and we developed this approach further in the present version of the INTERFACE concept, described above.

By contrast, the shared event approach partitions the variables among the machines, but allows machines to be coordinated by synchronising the execution of certain events in two machines at runtime. Obviously, this modifies the (meta level) scheduler that dispatches events. Furthermore, synchronised communication is achieved during synchronised execution by permitting output variables in one machine to be linked to input variables in another machine in order to transfer data at runtime. This approach is described in [12, 20].

In the world of continuous behaviours relevant to Hybrid Event-B, when a project consisting of several machines executes, each machine runs continuously. So in each machine, *some* event is always causing a transition (whether mode or pliant). In that sense, events are inevitably 'shared' in some manner in such a world. Additionally, since in general, there is no semantic reason to prevent the updates of variables from depending on the values of other variables (including variables shared via interfaces), variables also end up 'shared' in the relevant sense. So the multi-machine continuous world more or less forces the contemplation of both kinds of sharing simultaneously.

In our definition of multi-machine Hybrid Event-B we include both kinds of mechanism. Each is useful for specific modelling tasks, and there is no theoretical impediment to including both in the formal definition. Syntactic aspects are discussed in Section 7, while the semantics is set out in Section 11.

## 7. The Hybrid Event-B PROJECT

We now consider the Hybrid Event-B PROJECT, giving syntactic form to the ideas discussed above. Our definition specifies the minimum that enables the runtime semantics of Section 11 to be consistent, so at points below, we comment that a practical implementation may wish to be much more restrictive.

### *7.1. The PROJECT and its Constituents*

The PROJECT construct identifies all the constituent machines and interfaces of a project. If these come with all their variables and other names exactly tailored for cooperation, then that is enough. However, in genuine engineering contexts, we would also want to facilitate *reuse* of existing machines and interfaces, typically ones that were designed in a generic way to enable such reuse, so *construct instantiation* is another thing that we can delegate to the PROJECT construct. Coupled with this job is

```
PROJECT Example_Prj
  INTERFACE Itf1_IF
  INTERFACE Itf2_IF
  GLOBINVS Globals_GI
  MACHINE MA
  MACHINE MB IS
    CompMch WITH
      Cv1 → MBv1
      Cv2 → MBv2
      CExtV1 → Itf2v1
      CExtV2 → Itf2v3
      CExtV3 → Itf2v2
        •••
      Cev2.Cv! → MBev2.MBv!
  END
END
```

```
INTERFACE Itf1_IF
READS Itf2_IF
REFERS Itf2_IF
PLI/VAR Itf1v1,Itf1v2,Itf1v3
INVARIANTS
  Itf1inv1(Itf1v1,Itf1v2)
  Itf1inv2(Itf1v2,Itf1v3)
  Itf1tIIinv(Itf1v3,Itf2v1)
INITIALISATION
  •••
END
```

```
MACHINE MA
CONNECTS Itf1_IF
PLI/VAR MAv1,MAv2
INVARIANTS
  MAinv1(MAv1,MAv2)
  MAinv2(MAv1,MAv2)
EVENTS
  MAev1(Itf1v1,Itf1v2)
  MAev2(MAv1,Itf1v2,Itf1v3)
  MAev3(MAv1,MAv2)
END
```

```
INTERFACE Itf2_IF
READS Itf1_IF
REFERS Itf1_IF
PLI/VAR Itf2v1,Itf2v2,Itf2v3
INVARIANTS
  Itf2inv1(Itf2v1,Itf2v2,Itf2v3)
  Itf2inv2(Itf2v3)
  Itf2tIIinv(Itf2v1,Itf1v3)
INITIALSATION
  •••
END
```

```
MACHINE MB
CONNECTS Itf2_IF
PLI/VAR MBv1,MBv2
INVARIANTS
  MBinv1(MBv1,MBv2)
  MBinv2(MBv2)
EVENTS
  MBev1(MBv1,Itf2v1,Itf2v2)
  MBev2(MBv2,Itf2v3,MBv!)
END
```

```
MACHINE CompMch
PLI/VAR Cv1,Cv2,
  CExtV1,CExtV2,CExtV3
INVARIANTS
  Cinv1(Cv1,Cv2)
  Cinv2(Cv2)
EVENTS
  Cev1(Cv1,CExtV1,CExtV3)
  Cev2(Cv2,CExtV2,Cv!)
END
```

Figure 4: The syntax of a PROJECT by example, representing the machines and interfaces of Fig. 2.

the task of allowing such instantiated machines to communicate. For conceptual simplicity we would want such communication to have the semantics of shared variables.

We describe the project construct via examples, starting in Fig. 4 with the syntactic description of the system shown in Fig. 2.[4] Although the variables, invariants and events of the various constructs are not named in Fig. 2, for convenient reference we institute a uniform naming scheme as follows. Variables are named e.g. *MAv3*, the third variable in machine *MA* (counting left to right in *MA* in Fig. 2). Type I invariants are named e.g. *Itf2inv1*$(\ldots)$, the first invariant in *Itf2_IF* (counting left to right in *Itf2_IF* in Fig. 2) — the variables that occur in *Itf2inv1* are listed in parentheses. Events are named e.g. *MBev1*$(\ldots)$, the first event in *MB* (counting left to right in *MB* in Fig. 2) — again variables occurring in *MBev1* are listed in parentheses. An example of a type II invariant is *Itf2tIIinv*(*Itf2v1*, *Itf1v3*), labelled similarly.

On this basis, constructs *MA*, *MB*, *Itf1_IF*, *Itf2_IF* in Fig. 4 are easy to correlate with Fig. 2. The PROJECT construct *Example_Prj*, aside from listing these constructs, does two further things.

Firstly, via a declaration GLOBINVS *Globals_GI*, it allows the inclusion of a file (*Globals_GI*) of global invariants which are too complex to be subsumed under the tIi and tIIi patterns described earlier. Such invariants could involve the variables from any of the interfaces (but not machines) in the project, and we stipulate that *they should be derivable from the set of all the invariants contained in all the interfaces*. Thus, they would be verified when the interfaces were stable: either before more detailed work (thus making them requirements that must be established), or after detailed design of the project

[4]To save space, we compress the syntax of the machines. There is no difference between mode and pliant variables for these structural issues, so we declare them using PLI/VAR. Invariants and events simply list their variables in parentheses.

```
PROJECT ExampleR_Prj
REFINES Example_Prj
  INTERFACE Itf1_IF
  INTERFACE Itf2_IF
  MACHINE MM
  MACHINE MB
  SYNCH(MMsynch)
    MM.{MAev1,MMev4}
    MM.MAev3
  END
END
```

```
MACHINE MM
REFINES MA
CONNECTS Itf1_IF
PLI/VAR
  MMv3,MMv4,MMv5,MMv6
  MAv1,MAv2
INVARIANTS
  MMinv3(MMv3,MMv4)
  MMinv4(MMv4)
  MMinv5(MMv4,MMv5)
  MMinv6(MMv6)
  MMtIIinv7(MMv5,MAv1)
  MAinv1(MAv1,MAv2)
  MAinv2(MAv1,MAv2)
...  ...
```

```
...  ...
  EVENTS
    MAev1(MMv3,MMv4,
          Itf1v1,Itf1v2)
    MMev4(MMv4,MMv5)
    MMev5(MMv4,MMv6,MAv1)
    MAev2(MAv1,Itf1v2,Itf1v3)
    MAev3(MAv1,MAv2)
  END
```

Figure 5: A project for the refinement of the machines and interfaces of Fig. 2, as illustrated in Fig. 3.

was completed (making them confirmations of project-wide properties believed to be true).

Secondly, the project construct handles the instantiation of *MB* as an instance of machine *CompMch*. The clause *MB* IS *CompMch* does this. Part of the data of the clause is an injective mapping of *CompMch* names to *MB* names. This must preserve the attribute type of the names, i.e., mode and pliant state variables to mode and pliant state variables respectively, event names to event names, I/O bound variables to I/O bound variables etc.; and it must conform to the name space policy of Section 3.1. Variables outside the domain of the mapping are carried over unchanged. This notion of instantiation is rather basic, merely showing its feasibility. A more thoroughgoing notion for (discrete) Event-B is in [19].

While Fig. 4 shows the basic composition mechanism, Fig. 5 goes on to show the effects of refinement, based on the machines and interfaces shown in Fig. 3. Machine *MM* is a refinement of *MA*, which introduces four new variables *MMv3*-*MMv6*. These appear before the original *MAv1* and *MAv2* in *MM*. Also new are two events *MMev4* and *MMev5*, shown in *MM* appearing in between *MAev1* and *MAev2*. Finally there are new type I invariants *MMinv3*-*MMinv6* and the future type II invariant *MMtIIinv7*, again listed before the original invariants of *MA*. All are listed in left to right order (as they appear in the *MM* of Fig. 3) in the syntax of *MM* in Fig. 5.

## 7.2. Synchronised and Communicating Events

The last noteworthy element of Fig. 5 is the synchronisation clause SYNCH(*MMsynch*). This makes explicit the discussion of Section 6. It names the synchronisation *MMsynch*, and in its body, each line cites a collection of events (each being referred to in the form *MachineName*.*EventName*). This defines an AND/OR tree. Any execution of (any of the members of) that collection of events must contain the synchronised execution of exactly one event from each line. We assume that different synchronisations have no events in common.[5] In Fig. 5, this means that either *MAev1* or *MMev4* must be synchronised with *MAev3*, all being events of *MM*.

It may appear strange to synchronise events from the same machine, until we realise that (at least formally) the events of a collection of machines may need to reside in fewer larger machines prior to decomposition (see below), and that the synchronisation may be indispensable in maintaining crucial invariants.

[5]In practice, particularly when failures in multicomponent systems are being modelled, the simple synchronisation scheme just described can prove too inflexible. Various priority based enhancements could be envisaged to improve matters, but such considerations lie outside the scope of this paper.

The events participating in a synchronisation can communicate with each other via shared state variables like any other events, but this is not all. One property of synchronisations not visible in the syntax of projects thus far is that the ANY clauses of all events participating in a synchronisation form a single scope of bound parameters. Semantically, this means that any name occurring in multiple ANY clauses within a synchronisation is captured by all of those events, and refers to a single value (time dependent in the pliant case) across the whole synchronised family. As an aid towards ultimate implementation, such a shared local parameter family 'ANY *b*' may be rewritten into a single writer multiple readers family by decorating the single writer 'ANY *b*!' (with a suitable assignment to *b*! in the body of the event), and decorating the multiple readers 'ANY *b*?', with the relevant value bound in the WHERE guards. Evidently, such decorations must not clash with the input and output parameters for external I/O. An example occurs in Fig. 7, in the discussion of decomposition. However, the writer/readers formulation for this internal communication is just syntactic sugar, without separate semantics.

Note that synchronisations must be refined to synchronisations congruently, in order to conform to constraints **[♦16]**–**[♦17]** in Section 5. Specifically, if $Sync_A$ is refined to $Sync_C$, then every event $Ev_C$ that refines some event $Ev_A$ of $Sync_A$, must be present in $Sync_C$, also implying that the abstract to concrete refinement relationship for synchronisations is injective, as it is for machines.

We observe that in the scheme above, we synchronised arbitrary AND/OR combinations of events which are simple to deal with in the semantics. In practice though, simpler schemes are more practicable (contrast footnote 5). For example, restricting to binary synchronisations that implement unidirectional communication, or restricting to synchronisations of mode events only (as we will do in Section 13).

### *7.3. A Subtlety Concerning Edge-Triggered Phenomena*

In many comparable formalisms, successions of mode events (that immediately follow one another) are often permitted. Combined with the possibilities inherent in having collections of concurrent activities, considerable semantic complexity can arise (see e.g. [13] for a discussion of comparable issues in the context of Statecharts). In Hybrid Event-B similar avalanches of mode events are forbidden. However, this potentially constrains expressivity regarding edge-triggered phenomena, especially when coordination between edge-triggered phenomena in multiple machines is needed.

In multi-machine Hybrid Event-B, the approach to this is to use the semantics of direct assignment of pliant variables across families of synchronised events. The semantics of equality between time dependent functions, in particular of discontinuous functions, permits the immediate propagation of edge-triggered effects, without incurring the complexities arising from the analysis of the sequential dependencies of families of mode events.

## 8. Decomposition

The account so far permits us to assemble a large system by composing a number of machines and interfaces together. Of equal interest though for the B-Method (not to mention other methodologies), is the question of *decomposition* of a machine $M$ — $M$ may have grown too unwieldy through aggregation of design detail via refinement. Decomposing $M$ can enhance separate development of the various pieces.

Decomposition sounds like the inverse of composition, but it is not. When we compose several machines and interfaces, our aim is to enrich and thus to change the behaviour of the overall system, compared with what we might have had previously. But when we decompose a big system into smaller pieces, our aim is to *not* change the behaviour of the overall system, but just to change the way that the system is put together. The most immediate consequence of this in the Hybrid Event-B framework comes from the earlier noted fact that each machine is always executing *some* event. Thus if decomposition increases the number of machines, the amount of concurrency in the system increases too. Arranging for the behaviour of the overall system to *not* change then becomes non-trivial.[6]

Suppose machine $M$ is part of a project $\mathcal{P}$ (as defined above). We call its local variables the internal variables of $M$, and the variables contained in interfaces accessed by $M$, the external variables of $M$.

A major principle governing our decomposition scheme is that when $M$ is decomposed into a number of submachines $M_1 \ldots M_k$ and new interfaces $Itf_1 \ldots Itf_l$, then all other existing machines and interfaces of $\mathcal{P}$ remain unchanged (alternatives to this are possible of course). The aim is that the behaviour of the original project $\mathcal{P}$, and of the project $\mathcal{P}[_M\backslash^{M_1 \ldots M_k}]$ in which $M$ is replaced by $M_1 \ldots M_k$ and $Itf_1 \ldots Itf_l$ (with suitable adjustments to the PROJECT file) is the same. This amounts to equivalent behaviour of individual events in $M$ and in its decomposition, and to equivalent runtime scheduling choices in $\mathcal{P}$ and $\mathcal{P}[_M\backslash^{M_1 \ldots M_k}]$.

Consider a machine $M$ of $\mathcal{P}$, which we want to decompose. To simplify exposition, we decompose $M$ into just two machines, $MX$ and $MY$, with the help of a new interface *ItfXY_IF*. We consider pliant and mode events in turn.

Let *PliEv* be a pliant event of $M$. Since, when a project is running, each machine is executing some event, if *PliEv* is executing in $M$ in $\mathcal{P}$, the equivalent in $\mathcal{P}[_M\backslash^{M_1 \ldots M_k}]$ must involve pliant subevents executing in $MX$ and $MY$. Let these be *PliEvX* and *PliEvY* respectively. We deduce that: (a) *PliEvX* and *PliEvY* are always enabled simultaneously; (b) *PliEvX* and *PliEvY* simultaneously participate in the same scheduling choices in $\mathcal{P}[_M\backslash^{M_1 \ldots M_k}]$ that *PliEv* participates in in $\mathcal{P}$; (c) *PliEvX* and *PliEvY* define the same change of state in $\mathcal{P}[_M\backslash^{M_1 \ldots M_k}]$ that *PliEv* defines in $\mathcal{P}$.

Regarding (a), simultaneous enabledness, a synchronisation of *PliEvX* and *PliEvY* achieves it easily. However, just above, we suggested that in Section 13 we would restrict synchronisations to mode events only, so we reject this possibility. Another way to achieve (a) is to give *PliEvX* and *PliEvY* the same INIT and WHERE guards as *PliEv*, so this is what we stipulate. Consequently, all the variables that appear in the INIT and WHERE guards must be declared in *ItfXY_IF* (or in external interfaces).

Regarding (b), equivalent scheduling choices, suppose that in $\mathcal{P}$, at some scheduling point in some state of the system, *PliEv* and *PliEvOther* are both enabled. Then, the choice to schedule *PliEv* or *PliEvOther* is nondeterministic. In $\mathcal{P}[_M\backslash^{M_1 \ldots M_k}]$, at the equivalent point, *PliEvX* in *MX* and a corresponding part of *PliEvOther* in *MY* will both be enabled (because *PliEvX* and *PliEv* have the same guards; similarly for *PliEvOther* and *its* decomposition; and both *PliEv* and *PliEvOther* are enabled in the corresponding states). Thus it becomes impossible to prevent executing *PliEvX* and a part of *PliEvOther* simultaneously, violating the desired semantic equivalence between $\mathcal{P}$ and $\mathcal{P}[_M\backslash^{M_1 \ldots M_k}]$. Therefore, simultaneous satisfiability of the guards of distinct pliant events of $M$ completely blocks decomposition of $M$. To permit decomposition of $M$, we strengthen the preceding insight to insist that the WHERE guards of distinct pliant events are pairwise unsatisfiable, since the INIT guard of a pliant event may become false during its execution, although the event itself may remain eligible for continued execution, making the INIT guard an unreliable guide regarding the disabledness of its pliant event.

Regarding (c), same change of state in $\mathcal{P}$ and $\mathcal{P}[_M\backslash^{M_1 \ldots M_k}]$, we must arrange that the assigning clauses of *PliEv*, namely the COMPLY and SOLVE clauses, must be cleanly decomposed into corresponding pieces in *PliEvX* and *PliEvY*. Fig. 6 gives an illustrative example.

In Fig. 6, *PliEv* is shown with its COMPLY and SOLVE clauses already broken down into parts relevant to $x$ variables (which will become the focus of *PliEvX* in *MX*), and parts relevant to $y$ variables (which will become the focus of *PliEvY* in *MY*). Speaking generally, if the decomposition strategy

[6]This is in sharp contrast to the case of discrete Event-B in which events occur only at isolated moments. Then, there is no semantic difference between one machine not updating the state (except at isolated moments) and several machines not updating the state (except at isolated moments, and never simultaneously because of non-eager scheduling of events). In discrete Event-B, decomposition can in fact be viewed as a kind of inverse of composition.

```
PliEv
  STATUS pliant
  INIT $iv(x1,x2,y1,y2)$
  WHERE $grd(u)$
  ANY $i?$
  COMPLY
    $BDApredX(x1,x2,i?) \wedge$
    $BDApredY(y1,y2,i?)$
  SOLVE
    $\mathcal{D}\, x1 = \phi_X(x1,x2,u)$
    $\mathcal{D}\, y1 = \phi_Y(y1,y2,u)$
    $x2 := E_X(x1,u)$
    $y2 := E_Y(y1,u)$
  END
```

becomes

```
PliEvX
  STATUS pliant
  DECOMPOSES PliEv
  INIT $iv(x1,x2,y1,y2)$
  WHERE $grd(u)$
  ANY $i?$
  COMPLY
    $z = i? \wedge$
    $BDApredX(x1,x2,i?)$
  SOLVE
    $\mathcal{D}\, x1 = \phi_X(x1,x2,u)$
    $x2 := E_X(x1,u)$
  END
```

```
PliEvY
  STATUS pliant
  DECOMPOSES PliEv
  INIT $iv(x1,x2,y1,y2)$
  WHERE $grd(u)$
  COMPLY
    $BDApredY(y1,y2,z)$
  SOLVE
    $\mathcal{D}\, y1 = \phi_Y(y1,y2,u)$
    $y2 := E_Y(y1,u)$
  END
```

Figure 6: Decomposing a pliant event into smaller events.

follows the architectural structure of a system assembled out of physical components, this kind of clean separation is to be expected. As well as the copying of the guards of *PliEv* into those of *PliEvX* and *PliEvY*, and the partition of the assigning clauses, Fig. 6 shows what can be done if there is a single input $i?$ that is used in both parts of the prospective decomposition. We introduce a fresh state variable $z$ (declared and initialised in *ItfXY_IF*) to allow the value of $i?$ to be shared between $MX$ and $MY$. Of course, this idea of introducing intermediate state variables, is available more widely in principle, to decompose more complex constraints and assignments than we have discussed above. However, the calculations involving such variables can quickly become very complicated, so we do not pursue this line further.

Now let *MoEv* be a mode event of $M$. Unlike pliant events, we do not insist that mode events are executed simultaneously by all machines of a project at a given moment at runtime, creating a difference from the previous case. Therefore, mode events of $M$ may be allocated in their entirety to a submachine, or may be split into subevents *MoEvX* and *MoEvY*, as we had for pliant events.

We have the same concerns as previously: (a) simultaneous enabledness where needed; (b) equivalent scheduling choices; (c) same same change of state defined.

Regarding (a), simultaneous enabledness, if *MoEv* is not split there is no issue. If *MoEv* is split into *MoEvX* and *MoEvY* say, a synchronisation of *MoEvX* and *MoEvY* will ensure that they are executed simultaneously, giving a lot of flexibility in how the guard of *MoEv* is partitioned/duplicated in the guards of *MoEvX* and *MoEvY*. In the extreme case of complete duplication of the whole guard (as in the pliant case), the synchronisation may be omitted (unless required by (b), next).

Regarding (b), equivalent scheduling choices, the problems discussed for the pliant case persist, but we also have more remedies. Consider mode events of $M$ as follows, involving variables $x$ and $y$:[7]

*MoEv1* ... WHEN $x = 0 \wedge y = 0$ THEN $x := 1$ END
*MoEv2* ... WHEN $x = 0 \wedge y = 0$ THEN $y := 1$ END

In $M$, runtime scheduling nondeterminism forces mutual exclusion between *MoEv1* and *MoEv2*. If the guard becomes true at a certain moment, one or other would be selected to execute (assuming no further such events in $M$). Either choice disables both. But if, without any event splitting, *MoEv1* was placed in $MX$ and *MoEv2* was placed in $MY$, then, at the corresponding moment, both would be enabled, and according to the semantics of Section 11, both would execute, giving a semantics different from the $M$ semantics. This kind of thing must be prevented, which we do as follows.

[7]We are grateful to a reviewer for this counterexample.

```
MoEv
  STATUS ordinary
  ANY $a,b$
  WHERE $grdX(a,b) \wedge grdY(b)$
  THEN
    $x1 := E_X(\ldots)$
    $x2 := E_Y(\ldots)$
  END
```

becomes

```
MoEvX
  STATUS ordinary
  DECOMPOSES MoEv
  ANY $a,b!$
  THEN
    $x1 := E_X(\ldots)$
    $b! :| grdX(a,b!)$
  END
```

```
MoEvY
  STATUS ordinary
  DECOMPOSES MoEv
  ANY $b?$
  WHERE $grdY(b?)$
  THEN $x2 := E_Y(\ldots)$
  END
```

Figure 7: Decomposing a mode event into smaller events.

It is sufficient to insist that whenever two mode events *MoEv1* and *MoEv2* of $M$ can be simultaneously enabled in some state, then there is at least one submachine $M_j$ of the decomposition of $M$, such that the synchronised families of subevents (or the events themselves in the unsplit case) that decompose *MoEv1* and *MoEv2*, each contain a (sub)event declared in $M_j$. Such a constraint ensures that the runtime mutual exclusion between *MoEv1* and *MoEv2* forced by $M$ survives in the mutual exclusion between some (sub)events (of) *MoEv1* and *MoEv2* (and thus between the synchronisations that contain them) in $M_j$. We observe that the corresponding pliant event constraint (i.e. forbidding simultaneously enabled events) is just a simplification of this.

Regarding (c), same same change of state in $\mathcal{P}$ and $\mathcal{P}[_M\backslash^{M_1 \ldots M_k}]$, the measures we take are very similar to the pliant case, since the assignment syntax is the same. Again we illustrate with an example. Fig. 7 decomposes *MoEv* into the synchronised pair *MoEvX* and *MoEvY*. We see that the pattern for decomposing the assignments is exactly as in Fig. 6. Additionally, Fig. 7 shows what can be done if the original event *MoEv* introduces local parameters $a,b$ in its ANY clause. If these are to be used by more than one of the subevents of *MoEv*, then all such subevents must use the same values for their copies of $a,b$. As described in Section 7.2, we can use the local I/O capabilities of Hybrid Event-B to handle this. We enforce a single writer many readers discipline for such parameters, indicating the writer event by decorating its parameter with '!' and any reader events by decorating their parameters with '?'. In line with the view that output is an assigning operation, the *grdX* of *MoEv*, which assigned $a,b$ implicitly to satisfy the required constraints, has turned in *MoEvX*, into an explicit assignment of $b!$ to any value satisfying *grdX*. The semantics of such I/O is always that of instant propagation (of values among bound variables, as per Section 7.2).

We intend that the two ways of sharing local parameters illustrated for pliant and mode events respectively, namely: (1) the introduction of fresh shared state variables to make values available among subevents; and (2) the explicit use of I/O conventions and the single writer many readers discipline to make values available among subevents, should be viewed as being equally applicable to both kinds of event. Both mechanisms require a certain amount of lexical checking to prevent clashes. Thus in (1), ensuring that new variables are indeed fresh and remain so; and in (2), ensuring that parameters newly converted to I/O form do not clash with existing I/O parameters, and distinguishing newly introduced internal communications from pre-existing communication with the environment, are both required.

To recapitulate, if a machine $M$ is part of a project $\mathcal{P}$ and we wish to decompose it into submachines $M_1 \ldots M_k$ and new interfaces $Itf_1 \ldots Itf_l$, with events suitably decomposed as discussed above, the following must hold.

**[♦18]** Regarding the internal variables of $M$, the submachines and interfaces $M_1 \ldots M_k, Itf_1 \ldots Itf_l$ into which $M$ is decomposed conform to restrictions **[♦1]**–**[♦15]**.

**[♦19]** Any submachine $M_j$ that includes an event or subevent of $M$ that accesses an external variable of $M$, must access the relevant interface in the same manner that $M$ did.

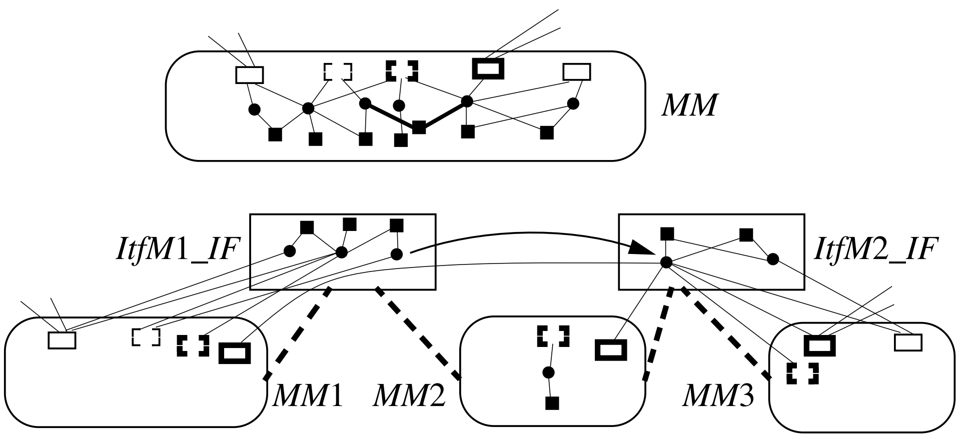

Figure 8: An illustration of the decomposition mechanism. *MA*, refined to a larger machine *MM*, is decomposed into smaller machines and interlinking interfaces. The trajectory of the individual elements through this process is described in the main text, and can also be discerned from the geometrical layout.

**[♦20]** Every pliant event *PliEv* of *M* is decomposed into subevents $PliEv_1 \dots PliEv_k$, one for each submachine $M_1 \dots M_k$, with each subevent having the same INIT and WHERE guards as *PliEv*, and with the assigning clauses appropriately distributed among the $PliEv_1 \dots PliEv_k$.

**[♦21]** For every pair of distinct pliant events of *M*, the (pairwise) conjunction of its WHERE guards is unsatisfiable.

**[♦22]** For every pair *MoEv1*,*MoEv2* of distinct mode events of *M*, neither of which has an input from the environment, if the (pairwise) conjunction of its WHERE guards is satisfiable, then there is a submachine $M_j$ of the decomposition of *M*, such that some subevent of the synchronised decomposition of *MoEv1* and some subevent of the synchronised decomposition of *MoEv2* (or, in each case, the event itself if the event is undecomposed), are both declared in $M_j$.

It is clear that adhering to **[♦18]** refines the partition of variables, when *M* is part of a larger project already adhering to **[♦1]**–**[♦15]**, thus not spoiling **[♦1]**–**[♦15]** overall.

Fig. 8 gives an example of decomposition at the project level, based on Fig. 3, where we have assumed machine *MA* has been refined to machine *MM* which has thereby been enlarged to the point where it needs to be split up. The syntax of the decomposition appears in Fig. 9. As already seen in Figs. 6 and 7 for events, we use a DECOMPOSES *Construct* clause in the syntax to indicate that the elements of the interface or machine in which it appears originate in *Construct*.

The lower part of Fig. 8 shows the decomposition of *MM* into *MM1*,*MM2*,*MM3*, and interfaces *ItfM1_IF*,*ItfM2_IF*. Although the same conventions as before determine the names of the various elements, we review this decomposition in detail because of its complexity.

Reading left to right in *MM*, across the top we have events *MAev1*,*MMev4*,*MMev5*,*MAev2*,*MAev3*. The events shown heavy, i.e. *MMev5*,*MAev2* are pliant, others are mode events. Across the middle we have variables *MMv3*,*MMv4*,*MMv5*,*MMv6*,*MAv1*,*MAv2*. Across the bottom we have invariants *MMinv3*,*MMinv4*,*MMinv5*,*MMinv6*,*MMtIIinv7*,*MAinv1*,*MAinv2*. This is all as in Fig. 5.

Below *MM* we have the interfaces of the decomposition. In *ItfM1_IF* we have the invariants *MMinv3*, *MMinv4*,*MMinv5* above the variables *MMv3*,*MMv4*,*MMv5*. Interface *ItfM2_IF* contains the invariants *MAinv1*,*MAinv2* above the variables *MAv1*,*MAv2*. Invariant *MMtIIinv7* has become a type II invariant straddling *ItfM1_IF* and *ItfM2_IF*.

```
PROJECT ExampleRD_Prj
DECOMPOSES ExampleR_Prj
  INTERFACE Itf1_IF
  INTERFACE Itf2_IF
  INTERFACE ItfM1_IF
  INTERFACE ItfM2_IF
  MACHINE MM1
  MACHINE MM2
  MACHINE MM3
  MACHINE MB
  SYNCH(MM12synch
    REFINES MMsynch)
    MM1.{MAev1,MMev4}
    MM3.MAev3
  END
  SYNCH(MMev5synch)
    MM1.MMev5_mm1
    MM2.MMev5_mm2
    MM3.MMev5_mm3
  END
  SYNCH(MAev2synch)
    MM1.MAev2_mm1
    MM2.MAev2_mm2
    MM3.MAev2_mm3
  END
END
```

```
INTERFACE ItfM1_IF
DECOMPOSES MM
READS ItfM2_IF
PLI/VAR MMv3,MMv4,MMv5
INVARIANTS
  MMinv3(MMv3,MMv4)
  MMinv4(MMv4)
  MMinv5(MMv4,MMv5)
  MMtIIinv7(MMv5,MAv1)
INITIALISATION
  •••
END
```

```
INTERFACE ItfM2_IF
DECOMPOSES MM
REFERS ItfM1_IF
PLI/VAR MAv1,MAv2
INVARIANTS
  MAinv1(MAv1,MAv2)
  MAinv2(MAv1,MAv2)
INITIALISATION
  •••
END
```

```
MACHINE MM1
DECOMPOSES MM
CONNECTS ItfM1_IF
EVENTS
  MAev1(MMv3,MMv4,
        Itf1v1,Itf1v2)
  MMev4(MMv4,MMv5)
  MMev5_mm1(MMv4)
  MAev2_mm1(MAv1)
END
```

```
MACHINE MM2
DECOMPOSES MM
CONNECTS ItfM1_IF
CONNECTS ItfM2_IF
PLI/VAR MMv6
INVARIANTS
  MMinv6(MMv6)
EVENTS
  MMev5_mm2(MMv6)
  MAev2_mm2(MAv1)
END
```

```
MACHINE MM3
DECOMPOSES MM
CONNECTS ItfM2_IF
EVENTS
  MAev2(MAv1,Itf1v2,Itf1v3)
  MAev3(MAv1,MAv2)
  MMev5_mm3(MAv1)
  MAev2_mm3(MAv1)
END
```

Figure 9: A project for the decomposition step of Fig. 8.

Below the interfaces, the machines *MM1*, *MM2*, *MM3* deal with the events. We assume that the mode events remain undecomposed, with the synchronisation between *MAev1* or *MMev4* and *MAev3* persisting, whereas the pliant events need to be decomposed and synchronised as described above. Thus *MM1* contains *MAev1*, *MMev4*, *MMev5_mm1*, *MAev2_mm1*. The latter two are the *MM1* shares of *MMev5*, *MAev2* from *MM*. Machine *MM1* contains events *MMev5_mm2*, *MAev2_mm2* which are the *MM2* shares of *MMev5*, *MAev2*. It also retains the local variable *MMv6* and local invariant *MMinv6* of *MM* as a local variable and local invariant of its own. Finally *MM3* contains *MMev5_mm3*, *MAev2_mm3*, *MAev3*, the first two being the *MM3* shares of *MMev5*, *MAev2*.

As we have seen, decomposition is intended to be a trivial kind of refinement. As an optimisation, and to avoid excessive duplication of models in a development, we can also introduce REFINESand-DECOMPOSES *Construct* clauses, which would combine a nontrivial refinement with a subsequent decomposition step. Although entirely within the spirit of our approach, the consequent syntactic constraints would become increasingly complex to check mechanically.

## 9. The Hypergraph Project Architecture Pattern

So far, we have presented a number of mechanisms that permit the construction of multi-machine projects in various ways, but we have not said anything about how these mechanisms should be used to best effect. Designers are, of course, entitled to use the features of multi-machine Hybrid Event-B as they see fit, but some general principles have emerged in case study work which are worth broadly recommending, and which can be labelled the *Hypergraph Project Architecture Pattern*.

If we consider each machine in a project as a node in a hypergraph, and each interface as a hyperedge joining all the nodes (machines) that connect to it, then a hypergraph structure emerges immediately. Alternatively, in the canonical reformulation of hypergraphs as bipartite graphs, we can view machine nodes and interface nodes as the two node kinds of a bipartite graph, and the CONNECTS and READS relationships between machines and interfaces as the bipartite graph's edges.

Given the combinatorial nature of the diamond rules, any legal Hybrid Event-B project gives rise to a hypergraph structure in this manner. However, the interface hyperedges may still be joined to one another via tIIi cross-cutting invariants. It is helpful to minimise the use of those in order to increase separate working. To do that we have to relate the hypergraph structure to the problem domain.

Normally, at least at a sufficiently low level of abstraction, the machines of a project will correspond to actual system components. Accepting this in principle, the issue remains of how to partition all the variables into interfaces.

At one extreme, we can put all the variables that will be updated by a machine into a separate interface conceptually belonging to that machine — although there is no formal requirement that variables are updated by unique machines in Hybrid Event-B, this very often happens in practice at low levels of abstraction. Partitioning the variables this way typically gives rise to tIIi's, as other machines need read access to the variables, while being coupled to the variables' owning machines by nontrivial invariants.

At the other extreme, we can analyse the responsibilities of the various machines, and partition the variables according to the separate concerns that have to be taken care of within the project. It is often the case that the invariants that are judged important from an applications perspective align well with the various concerns, and often, a good separation is seen between the sets of variables involved in the invariants belonging to different concerns. This minimises or eliminates the need for tIIi's.

The recommendation is to follow the second route. This gives the maximum flexibility for writing important complex invariants, which may couple different machines' events nontrivially, yet because they only involve the variables of a single interface, need not be restricted to the tIIi pattern. This recommendation is called the *Hypergraph Project Architecture Pattern*.

The main evidence for this pattern comes from case study work. In [11] a significant landing gear case study problem is proposed. In [7] there is a Hybrid Event-B development of it, which, due to early modelling decisions, in effect followed the first route. This gave rise to the need for a number of tIIi's distributed around the various interfaces. The case study was revisited and extended in [8], and this time the earlier modelling decisions were overturned and a hypergraph architecture as we recommend here emerged naturally. It proved possible to eliminate all use of tIIi's thereby. Besides this, in [10], a PID controller case study for yaw control of an e-vehicle is developed using the hypergraph architecture pattern, and confirms its good structural properties.

## 10. Small PROJECT Case Studies

In this section we present two small project case studies, small by necessity, since genuine multi-component multistage developments would be too big to include. Larger case studies have already been mentioned: [7, 8, 10]. The case studies here are restricted to a small development done in discrete Event-B in Section 10.1, and, in Section 10.2, there is a reexamination of the European Train Control System, first examined using Hybrid Event-B as a single machine in PaperI. We present these at this point since they are primarily concerned with illustrating the structural mechanisms, and do not require the detailed semantic considerations that occupy the remainder of the paper.

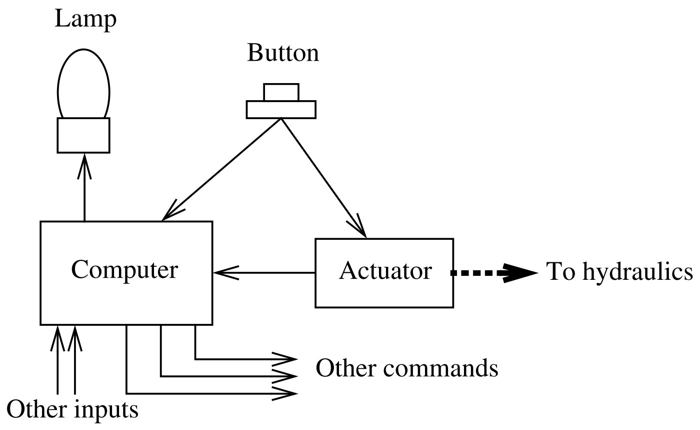

Figure 10: A simplified high power switchgear switching mechanism.

## *10.1. Power Switching*

In this section we consider a fragment of a power grid application. The fragment concerns the switching in or out of the grid, of high current high voltage circuits. The switches that manage such transitions are complex pieces of equipment. Fig. 10 shows our system. There is a button, pressed by the operator, to bring in a circuit.

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 hydraulic actuator to power up the hydraulics that will cause the movement of the electromechanical components. Successful powerup of the hydraulics is signalled to the computer from a sensor in the hydraulic actuator. This signal acts as a safety interlock that helps prevent malfunction of the system (which could result in costly damage to the switchgear). 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. Fig. 11 contains a top level model of the system. We use discrete Event-B here since our models do not require any nontrivial continuous behaviour.

```
PROJECT PowerSw0_Prj
  MACHINE Oper
END
```

```
MACHINE Oper
VARIABLES button, lamp
INVARIANTS
  button ∈ {UP, DOWN}
  lamp ∈ {OFF, ON}
  lamp = ON ⇒ button = DOWN
EVENTS
  INITIALISATION
    BEGIN
      button, lamp := UP, OFF
    END
... ...
```

```
... ...
    ButtonDown
      WHEN button = UP
      THEN button := DOWN
      END
    LampOn
      WHEN button = DOWN
      THEN lamp := ON
      END
  END
```

Figure 11: Top level (operator's) model of the power switching system.

```
  PROJECT PowerSw1_Prj
  REFINES PowerSw0_Prj
    MACHINE FullOperR
  END
```

```
  MACHINE FullOperR
  REFINES Oper
  VARIABLES
    button, buttonC, buttonH,
    hydraulics, lampsignal, lamp
  INVARIANTS
    $button \in \{UP, DOWN\}$
    $buttonC \in \{UP, DOWN\}$
    $buttonH \in \{UP, DOWN\}$
    $buttonC = DOWN \Rightarrow button = DOWN$
    $buttonH = DOWN \Rightarrow button = DOWN$
    $lampsignal \in \{OFF, ON\}$
    $lamp \in \{OFF, ON\}$
    $lampsignal = ON \Rightarrow$
          $buttonC = DOWN \land hydraulics = ON$
    $lamp = ON \Rightarrow lampsignal = ON$
    $lamp = ON \Rightarrow button = DOWN$
    $hydraulics \in \{OFF, ON\}$
    $hydraulics = ON \Rightarrow buttonH = DOWN$
  EVENTS
    INITIALISATION
      BEGIN
        button, buttonC, buttonH := UP, UP, UP
        hydraulics := OFF
        lampsignal, lamp := OFF, OFF
      END
... ...
```

```
... ...
    ButtonDown
      REFINES ButtonDown
      WHEN $button = UP$
      THEN button := DOWN
      END
    ButtonDownC
      WHEN $button = DOWN$
      THEN buttonC := DOWN
      END
    ButtonDownH
      WHEN $button = DOWN$
      THEN buttonH := DOWN
      END
    HydraulicsOn
      WHEN $buttonH = DOWN$
      THEN hydraulics := ON
      END
    LampSignal
      WHEN $buttonC = DOWN \land hydraulics = ON$
      THEN lampsignal := ON
      END
    LampOn
      REFINES LampOn
      WHEN $lampsignal = ON$
      THEN lamp := ON
      END
  END
```

Figure 12: Enriched top level model, ready for decomposition.

Our aim is to refine and decompose this top level view to model the different components separately, and to add the sensor behaviour mentioned. As a first step, we refine the machine *Oper* to a machine *FullOperR* that includes all the pieces needed for the decomposition, so that the decomposition itself becomes the simple rearrangement we described in Section 8. The result is shown in Fig. 12, where all the detail has been built into the machine *FullOperR*. This has an additional two copies of the *button* variable, one each for the upcoming *Comp* and *Hydraulics* machines. The original events of *Oper* are present, along with some new events. Note that the PROJECT files have had little to do yet.

In Fig. 13 we decompose *FullOperR* into *OperR*, *Comp* and *Hydraulics*. The *OperR* machine represents the operator's view in the decomposed system. The *Comp* machine is intended as the head of a refinement development, to take into account an increasing number of sensor inputs and actuator outputs as the development progresses. The *Hydraulics* machine is also intended to become the head of a refinement development, to take into account more detail in the hydraulic system.

This small system illustrates our decomposition method rather well. Control starts with the operator, and is then passed to both the computer and hydraulics subsystems via the button press. After some activity in the two separate subsystems, reinforcement of normal functioning in both subsystems is gained by the sending of the $hydraulics = ON$ signal from the hydraulics subsystem, and its correct processing by the computer subsystem. As already noted, this kind of mutual confirmation is rather common in critical systems. However, the to-and-fro of control and information that is needed for this is typically

```
PROJECT PowerSw2_Prj
DECOMPOSES PowerSw1_Prj
  INTERFACE Comp_IF
  INTERFACE Hydr_IF
  MACHINE OperR
  MACHINE Comp
  MACHINE Hydraulics
END
```

```
INTERFACE Comp_IF
DECOMPOSES FullOperR
READS Hydr_IF
REFERS Hydr_IF
VARIABLES $button, buttonC, lampsignal, lamp$
INVARIANTS
  $button \in \{UP, DOWN\}$
  $buttonC \in \{UP, DOWN\}$
  $buttonC = DOWN \Rightarrow button = DOWN$
  $lampsignal \in \{OFF, ON\}$
  $lamp \in \{OFF, ON\}$
  $lampsignal = ON \Rightarrow$
      $buttonC = DOWN \wedge hydraulics = ON$    $(*1)$
  $lamp = ON \Rightarrow lampsignal = ON$
  $lamp = ON \Rightarrow button = DOWN$
INITIALISATION
  BEGIN
    $button, buttonC := UP, UP$
    $lampsignal, lamp := OFF, OFF$
  END
END
```

```
INTERFACE Hydr_IF
DECOMPOSES FullOperR
READS Comp_IF
REFERS Comp_IF
VARIABLES $buttonH, hydraulics$
INVARIANTS
  $buttonH \in \{UP, DOWN\}$
  $hydraulics \in \{OFF, ON\}$
  $buttonH = DOWN \Rightarrow button = DOWN$    $(*2)$
  $hydraulics = ON \Rightarrow buttonH = DOWN$
INITIALISATION
  BEGIN
    $buttonH := UP$
    $hydraulics := OFF$
  END
END
```

```
MACHINE OperR
DECOMPOSES FullOperR
CONNECTS Comp_IF
EVENTS
  ButtonDown
    WHEN $button = UP$
    THEN $button := DOWN$
    END
  LampOn
    WHEN $lampsignal = ON$
    THEN $lamp := ON$
    END
END
```

```
MACHINE Comp
DECOMPOSES FullOperR
CONNECTS Comp_IF
EVENTS
  ButtonDownC
    WHEN $button = DOWN$
    THEN $buttonC := DOWN$
    END
  LampSignal
    WHEN $buttonC = DOWN \wedge hydraulics = ON$
    THEN $lampsignal := ON$
    END
END
```

```
MACHINE Hydraulics
DECOMPOSES FullOperR
CONNECTS Hydr_IF
EVENTS
  ButtonDownH
    WHEN $button = DOWN$
    THEN $buttonH := DOWN$
    END
  HydraulicsOn
    WHEN $buttonH = DOWN$
    THEN $hydraulics := ON$
    END
END
```

Figure 13: The decomposed power switching development.

handled rather poorly in traditional model based decomposition techniques, which are normally much more oriented towards a pure divide-and-conquer strategy.

Our decomposition creates interfaces *Comp_IF* and *Hydr_IF*. The former is primarily concerned with the *Comp* machine and the latter with the *Hydraulics* machine. The invariants linking computation and hydraulics cross-cut this structure, which we take care of using the tIIi's marked $(*1)$ and $(*2)$ in Fig. 13.

```
INTERFACE HydrR_IF
REFINES Hydr_IF
READS Comp_IF
REFERS Comp_IF
VARIABLES buttonH, hydraulics, pump
INVARIANTS
  buttonH ∈ {UP, DOWN}
  hydraulics ∈ {OFF, ON}
  pump ∈ {OFF, ON}
  buttonH = DOWN ⇒ button = DOWN    (∗2)
  hydraulics = ON ⇒ pump = ON
INITIALISATION
  BEGIN
    buttonH, hydraulics, pump := UP, OFF, OFF
  END
END
```

```
MACHINE HydraulicsR
REFINES Hydraulics
CONNECTS HydrR_IF
VARIABLES pumpfail
INVARIANTS pumpfail ∈ BOOL
EVENTS
  INITIALISATION
    BEGIN pumpfail := FALSE END
  ButtonDownH
    WHEN button = DOWN
    THEN buttonH := DOWN
    END
  Pumpfail
    WHEN ¬pumpfail
    THEN pumpfail := TRUE
    END
  PumpOn
    WHEN buttonH = DOWN ∧ ¬pumpfail
    THEN pump := ON
    END
  HydraulicsOn
    WHEN pump = ON
    THEN hydraulics := ON
    END
END
```

Figure 14: The further refined hydraulic system.

The tIIi $(*2)$ tracks the waking of control in the hydraulic subsystem when the operator button is pressed, while tIIi $(*1)$ records the return of control to the computer subsystem once the hydraulics have done enough of their task, the computer having also been busy with its own tasks in the meantime. At a global level we have not lost track of the dependencies between all the events, while nevertheless facilitating separate development. However, this structure does not follow our recommended Hypergraph Architecture Pattern, because the variables have been aggregated into interfaces according to the machines that update them, rather than according to their functional concern.

We consider the addition of a small amount of extra detail to our system, refining it by including the engaging and monitoring of the pump that pressurises the hydraulic circuit that will drive the switching gear. We give just the refined fragment, rather than including the whole project. This appears in Fig. 14, which contains the refined interface *HydrR_IF* and the refined machine *HydraulicsR*. The latter introduces the pump, and also, a failure mode for the pump, which when set, prevents the *hydraulics* variable from being set, reflecting the system characteristics discussed earlier.

Superficially, Figs. 13 and 13 contain a violation of our decomposition rules, in that there are two events in machine *FullOperR*, namely *ButtonDownC* and *ButtonDownH*, both of which are enabled when $button = DOWN$, and which are put into different submachines *Comp* and *Hydraulics* during decomposition. We make several observations about this.

First observation. This is a discrete Event-B development. So there are no pliant events and all (mode) events execute lazily, temporally isolated from each other, the only coincidences in execution times being those forced via synchronisations. Therefore the measures taken to prevent undesired scheduling possibilities are not needed. Specifically **[♦20]**–**[♦22]** are not needed for discrete Event-B, and the structural rules alone **[♦18]**–**[♦19]** are sufficient. Since **[♦22]** does not apply and *ButtonDownC* and *ButtonDownH* cannot execute simultaneously, there is no violation.

Second observation. The discrete Event-B development may be regarded as a shorthand for a corresponding Hybrid Event-B development whose non-trivial events are all mode events. Such a development may be translated to legitmate Hybrid Event-B by adding a COMPLY *INVARIANTS* default pliant event, and adding to each mode event an (otherwise redundant) input from the environment to ensure lazy execution. Now **[♦20]**–**[♦22]** are applicable, and although the events execute lazily, they execute in different machines, and the scheduling policy of Hybrid Event-B would not prevent them being scheduled simultaneously (this follows from points **[10]** and **[12.1]** in the semantics of Section 11). So there is a violation.

Third observation. Following on from the preceding, we could easily add a boolean variable *turn* to machine *FullOperR* to artificially create sequentialisation, setting it nondeterministically in *ButtonDown* say, and guarding *ButtonDownC* and *ButtonDownH* on opposite values of *turn*, negating its value in each of these events. This cures the violation. During decomposition, if we put *turn* in *Comp_IF*, then machine *Hydraulics* would need an additional CONNECTS *Comp_IF* to access *turn*.

Fourth observation. As an alternative to the previous case, we notice that even when *ButtonDownC* and *ButtonDownH* are executed simultaneously, there is no detrimental effect on the overall system behaviour since the updates are non-interfering and both are needed for the ultimate goal of switching the lamp on. This demonstrates that **[♦22]** is certainly very conservative. More liberal variations on condition **[♦22]** can be imagined, but these would entail substantial excursions into static analysis, which we wish to avoid in this paper.

Fifth observation. The mode event scheduling policy of Hybrid Event-B mentioned above (points **[10]** and **[12.1]** of the semantics) is quite specific. Although it prevents a mode event with input from executing immediately after a preceding mode event, it does not prevent more than one mode event with input from executing simultaneously (or indeed simultaneously with other mode events without input) provided all belong to different machines. The motivation is connected with refinement. In Hybrid Event-B, since default event scheduling is eager, inputs from the environment are used to introduce elements of nondeterminism. Since refinement typically reduces nondeterminism, environmental inputs may, during refinement, be replaced by more deterministic and more eager behaviour modelled using state variables. If inputs were always constrained to execute apart from each other and apart from non-input events (which could easily be arranged), some refinements of this kind could be rendered impossible, given that time runs at the same rate in all levels of a refinement chain in Hybrid Event-B.

## *10.2. European Train Control System*

In this example, we reconsider the treatment of the European Train Control System (ETCS) from PaperI, in the light of the multi-machine theory of this paper. Since there are two obvious agents in the ETCS model, the *Radio Block Controller* and the *Train*, having a machine for each is evidently more desirable than a single machine treatment, all other things being equal.

In the ETCS, the rail track is organised into dynamically controlled **movement authorities**. The key invariants are that **distinct movement authorities are always disjoint**, that **each movement authority contains (at most) one train**, and that **each train is in some movement authority**. If these are always maintained, then trains cannot collide.

Since the present treatment is just a repackaging of the PaperI treatment, we do not repeat the arguments that justify the mathematical details of the model. Instead, we focus on the various different pieces, and how they fit together. Accordingly, we go straight to the generalised movement authority model of Fig. 15. In this, we have train dynamical variables, $\tau.p$, $\tau.v$ and $\tau.a$ which represent the current position, velocity and acceleration of the train, respectively, and the train emergency braking distance $\tau.sb$. We also have movement authority variables $\mathbf{m}.r$, $\mathbf{m}.e$ and $\mathbf{m}.d$, representing respectively the *recommended* speed (away from the emergency braking zone), the movement authority *endpoint*, and the *demanded* maximum speed permissible at the endpoint. The key invariant is thus $\tau.p \geq \mathbf{m}.e \Rightarrow \tau.v \leq \mathbf{m}.d$.

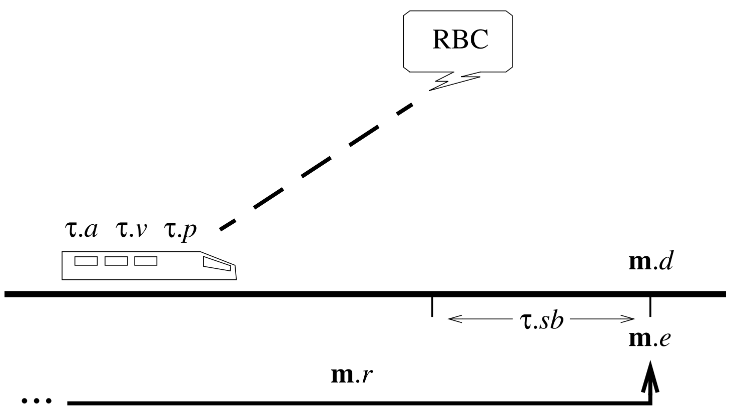

Figure 15: The European Train Control System. A generalised movement authority, defined by its recommended speed limit $\mathbf{m}.r$, end position $\mathbf{m}.e$ and demanded speed limit (at end) $\mathbf{m}.d$. This is used to control the train parameters: train acceleration $\tau.a$, train speed $\tau.v$, and train position $\tau.p$. The essential safety invariant is $\tau.p \geq \mathbf{m}.e \Rightarrow \tau.v \leq \mathbf{m}.d$.

## *10.3. A Hybrid Event-B Project for ETCS*

The structural part of the ETCS Hybrid Event-B Project is shown in Fig. 16, containing the PROJECT *ETCS_Prj* file itself, the CONTEXT *ETCS_Ctx* static data file, and the INTERFACE *ETCS_IF* file.

The project file identifies the component constructs contributing to the project, namely the context file, the interface file, and the *RadioBlockController* and *TrainController* machines. It also contains the synchronisation specifications, which are very simple in this project; all that is needed is to specify that the *MOVEMENT_AUTHORITY* mode events in the two machines are to execute simultaneously (these being a decomposition of a single event of PaperI into two).

The context file contains the *normal* and *emergency* mode constants, and the emergency brake *emrg* and new movement authority *newMA* message values. It also contains the maximum train deceleration and acceleration, $b$ and $A$, also $\varepsilon$, which is the polling interval. In addition, it contains two static functions, $\mathsf{bd}$ and $\mathsf{od}$ (braking distance and overshoot distance) needed in the safety calculations.

The interface file contains almost all of the variables of the system. Many of them are accessed by both machines. The train position and velocity $\tau.p$ and $\tau.v$ are pliant because they are required to change continuously when following a physical law. The other dynamical variables (train acceleration and movement authority data) are only changed by mode events, so their behaviour will be piecewise constant (albeit that their values are drawn from $\mathbb{R}$), so we can declare them as mode variables. A perusal of the code of the two machines reveals that each variable is only updated by a single machine of the project, even if it is accessed by both, which accords with static criteria for correctness discussed below. Since there is only one interface file, the project has a hypergraph architecture automatically.

The key safety property appears as *inv9* in Fig. 16.[8] Observe that if all the train variables $\tau.\_$ lived in the *TrainController* (train) machine, and all the movement authority variables $\mathbf{m}.\_$ lived in the *RadioBlockController* (RBC) machine, then if we put *inv9* into the RBC machine, then we would have to existentially quantify the train variables in *inv*9. While it is one thing to say that values for the train variables *exist* that make *inv9* true, it is quite another to say that the *current train variable values* do so. A corresponding problem arises if *inv9* were put into the train machine — and doing both of these things is no good either, since neither set of existential witnesses would be connected to the actual system values at any particular time. Inspecting the details of the train machine and the RBC machine reveals

[8]The labels of the invariants are consistent with PaperI.

```
PROJECT ETCS_Prj
  CONTEXT ETCS_Ctx
  INTERFACE ETCS_IF
  MACHINE RadioBlockController
  MACHINE TrainController
  SYNCHRONISE
    RadioBlockController.
      {MOVEMENT_AUTHORITY}
    TrainController.
      {MOVEMENT_AUTHORITY}
  END
END
```

```
CONTEXT ETCS_Ctx
  SETS MODES, MSGS
  CONSTANTS
    normal, emergency
    emrg, newMA
    $b, A, \varepsilon$
    bd, od
  AXIOMS
    $MODES =$
      $\{normal, emergency\}$
    $MSGS = \{emrg, newMA\}$
    $b \in \mathbb{R} \wedge b > 0$
    $A \in \mathbb{R} \wedge A > 0$
    $\varepsilon \in \mathbb{R} \wedge \varepsilon > 0$
    $\mathsf{bd} \in \mathbb{R} \times \mathbb{R} \to \mathbb{R}$
    $\forall x, y \bullet \mathsf{bd}(x, y) = \frac{x^2 - y^2}{2 \times b}$
    $\mathsf{od} \in \mathbb{R} \to \mathbb{R}$
    $\forall z \bullet \mathsf{od}(z) = z\varepsilon + \frac{1}{2} A\varepsilon^2$
END
```

```
INTERFACE ETCS_IF
SEES ETCS_Ctx
PLIANT
  $\tau.p, \tau.v$
VARIABLES
  $mode, \mathbf{m}.r, \mathbf{m}.e, \mathbf{m}.d$
INVARIANTS
  $inv0 : \tau.p \in \mathbb{R} \wedge \tau.p \geq 0$
  $inv1 : \tau.v \in \mathbb{R} \wedge \tau.v \geq 0$
  $inv2 : mode \in MODES$
  $inv5 : \mathbf{m}.r \in \mathbb{R} \wedge \mathbf{m}.r \geq 0$
  $inv6 : \mathbf{m}.e \in \mathbb{R} \wedge \mathbf{m}.e \geq 0$
  $inv7 : \mathbf{m}.d \in \mathbb{R} \wedge \mathbf{m}.d \geq 0$
  $inv8 : \mathbf{m}.r \geq \mathbf{m}.d$
  $inv9 : \tau.p \geq \mathbf{m}.e \Rightarrow \tau.v \leq \mathbf{m}.d$
INITIALISATION
  BEGIN
    $\tau.p, \tau.v := 0, 0$
    $mode := normal$
    $\mathbf{m}.r, \mathbf{m}.e, \mathbf{m}.d := 0, 0, 0$
  END
END
```

Figure 16: Project construct, static data, and shared variables and invariants for ETCS.

that train variables $\tau.\_$ are only inspected and updated in the train machine. Therefore the *only* reason for putting them in the interface *ETCS_IF* is the need to relate them to the movement authority variables in the critical safety invariant *inv9*.

We now consider the behaviour of the system, though briefly, since it is the same as in PaperI. The *RadioBlockController* machine in Fig. 17, after initialising, engages in the *IDLE_RBC* pliant event, which does nothing, waiting to be interrupted either by the *MOVEMENT_AUTHORITY* mode event (which adjusts the movement authority variables to new values), or by the *EMERGENCY* mode event (which notifies an emergency ahead). Both mode events have input parameters, so, according to the semantics, the needed values become available at undetermined times that do not clash with any other mode event occurrences. Note that *EMERGENCY* can only occur once, since it disables its own guard, after which it brings the system to rest.

Note the specifications of the pliant behaviour in the *IDLE_RBC* and *FINAL_RBC* events. These say COMPLY CONST(...). This utilises the CONST pliant modality [5] demanding that the values of the variables mentioned do not change during the pliant transition. This makes clear which variables (mode or pliant) are being controlled by the event, but is strictly speaking superfluous. Equally good would be COMPLY *Invariants*, the usual default, applying to all the variables controlled by the machine, which would work here, since all the variables mentioned are mode variables and thus are forbidden from changing during a pliant event in the machine that controls them. Also permissible would be COMPLY skip which says 'do nothing' to the values of the variables controlled by the machine during the pliant event, again consistent with the expected constant behaviour of all the relevant variables.

Focusing on the *MOVEMENT_AUTHORITY* event, when prompted by the *newMA* message from the environment, it reassigns the movement authority variables according to spontaneously generated output parameter values $r!, e!, d!$. These are passed to the train during the simultaneous execution of the train's *MOVEMENT_AUTHORITY* event. (Note that the formal semantics of this is the spontaneous generation of values for shared bound variables $r, d, e$, despite the I/O appearance of the variable names.) In PaperI, we gave a careful justification of why the particular restrictions placed on the various parameters were

```
MACHINE  RadioBlockController
CONNECTS  ETCS_IF
EVENTS
  IDLE_RBC
    STATUS  pliant
    WHEN  $mode = normal$
    COMPLY
      CONST($mode$,
        $\mathbf{m}.r, \mathbf{m}.e, \mathbf{m}.d$)
    END
...  ...
```

```
...  ...
  MOVEMENT_AUTHORITY
    STATUS  ordinary
    ANY  $msg?, r!, e!, d!$
    WHERE
      $mode = normal \wedge$
      $msg? = newMA \wedge$
      $r! \in \mathbb{R} \wedge r! > 0 \wedge$
      $e! \in \mathbb{R} \wedge e! > 0 \wedge$
      $d! \in \mathbb{R} \wedge d! > 0 \wedge$
      $r! \geq d! \wedge$
      $(d! \geq \mathbf{m}.d \Rightarrow e! \geq \mathbf{m}.e) \wedge$
      $(d! \leq \mathbf{m}.d \Rightarrow$
          $e! \geq \mathbf{m}.e + (\mathbf{m}.d^2 - d!^2)/2b)$
    THEN
      $\mathbf{m}.r, \mathbf{m}.e, \mathbf{m}.d$ := $r!, e!, d!$
    END
...  ...
```

```
...  ...
  EMERGENCY
    STATUS  ordinary
    ANY  $msg?$
    WHERE
      $mode = normal \wedge$
      $msg? = emrg$
    THEN
      $mode$ := $emergency$
    END
  FINAL_RBC
    STATUS  pliant final
    WHEN
      $mode = emergency$
    COMPLY
      CONST($mode$,
        $\mathbf{m}.r, \mathbf{m}.e, \mathbf{m}.d$)
    END
END
```

Figure 17: The *RadioBlockController* machine for the European Train Control System.

appropriate. We do not repeat that here.

We turn to the *TrainController* machine, shown in Fig. 18. This also CONNECTS to the INTERFACE *ETCS_IF*. It declares a train internal clock $\tau.clk$ to manage the train's polling behaviour, and also the train acceleration variable $\tau.a$, which ranges between $-b$ (maximum deceleration) and $A$ (maximum acceleration).

The only non-final pliant event is *DRIVE*, scheduled whenever the clock is reset to 0, lasting for a period $\tau.clk < \varepsilon$. At the reset, various mode events can become enabled (via a guard $\tau.clk = \varepsilon$), thus continuing the system trace. The *DRIVE* event just stipulates Newtonian mechanics.

Of the mode events, *MOVEMENT_AUTHORITY* is the most complex, performing the train's part of the synchronised movement authority update. As before, we refer to PaperI for the technical details. Mode events *SPEED_OK* and *SPEED_HIGH* manipulate the train's speed during normal operation.

If, by the end of a polling interval, the mode has been set to *emergency* by the *RBC*, or the emergency braking zone has been entered, then in *AUTOMATIC_TRAIN_PROTECTION*, the acceleration is set to maximum braking and the clock is reset. The last mode event, *FULL_STOP*, is triggered when the velocity reaches 0 during emergency braking, whereupon the train's motion stops, enabling the final pliant event *FINAL_TRAIN*.

The brief outline of the ETCS system just given is to be compared with the more detailed single machine treatment in PaperI. Although the specific formulae were carefully justified there, the point here is that the multi-machine formulation is essentially just a splitting up of the single machine version. In that sense, it illustrates well our objective in this paper that decomposition should be little more than a syntactic partitioning process, and that the semantics of a decomposed system should be readily understandable in terms of a recomposed monolithic system.

## 11. Formal Semantics

The formal semantics of a single Hybrid Event-B machine is discussed in detail in PaperI. The formal semantics of multi-machine projects is essentially the same, but with two complications. The first is the fact that we allow events to be synchronised (usually across more than one machine, but not necessarily, allowing for cases like *MMsynch* in Fig. 5). This implies that the 'guard at runtime' of such a collection

```
MACHINE TrainController
CONNECTS ETCS_IF
CLOCK $\tau.clk$
VARIABLES $\tau.a, \tau.sb$
INVARIANTS
  $inv2 : \tau.a \in \mathbb{R} \wedge$
        $\tau.a \in [-b \ldots A]$
  $inv3 : \tau.sb \in \mathbb{R} \wedge$
        $\tau.sb \geq 0$
EVENTS
  INITIALISATION
    STATUS ordinary
    BEGIN
      $\tau.a := 0$
      $\tau.sb := 0$
      $\tau.clk := 0$
    END
  DRIVE
    STATUS pliant
    WHEN $\tau.clk = 0$
    COMPLY $\tau.v \geq 0 \wedge$
           $\tau.clk < \varepsilon$
    SOLVE $\mathcal{D}\tau.v = \tau.a$
          $\mathcal{D}\tau.p = \tau.v$
    END
  ...  ...
```

```
...  ...
  MOVEMENT_AUTHORITY
    STATUS ordinary
    WHEN $mode = normal$
    ANY $r?, e?, d?$
    WHERE
      $r? \in \mathbb{R} \wedge e? \in \mathbb{R} \wedge d? \in \mathbb{R}$
    THEN
      $\tau.sb :=$
        $\mathsf{bd}(r? + A\varepsilon, d?) + \mathsf{od}(r?)$
    END
  SPEED_OK
    STATUS ordinary
    ANY $a$
    WHEN
      $\tau.clk = \varepsilon \wedge$
      $mode = normal \wedge$
      $\mathbf{m}.e - \tau.p \geq \tau.sb \wedge$
      $\tau.v \leq \mathbf{m}.r \wedge$
      $a \in [-b \ldots A]$
    THEN $\tau.a := a$
         $\tau.clk := 0$
    END
  ...  ...
```

```
  ...  ...
  SPEED_HIGH
    STATUS ordinary
    WHERE
      $\tau.clk = \varepsilon \wedge mode = normal \wedge$
      $\mathbf{m}.e - \tau.p \geq \tau.sb \wedge \tau.v \geq \mathbf{m}.r$
    THEN $\tau.a := -b$
         $\tau.clk := 0$
    END
  AUTOMATIC_TRAIN_PROTECTION
    STATUS ordinary
    WHEN
      $\tau.clk = \varepsilon \wedge$
        $(mode = emergency \vee$
        $\mathbf{m}.e - \tau.p \leq \tau.sb)$
    THEN $\tau.a := -b$
         $\tau.clk := 0$
    END
  FULL_STOP
    STATUS ordinary
    WHEN
      $\tau.v = 0 \wedge mode = emergency$
    THEN $\tau.a := 0$
         $\tau.clk := 0$
    END
  FINAL_TRAIN
    STATUS pliant final
    WHEN
      $\tau.clk = 0 \wedge mode = emergency \wedge$
      $\tau.v = 0 \wedge \tau.a = 0$
    COMPLY skip
    END
END
```

Figure 18: The *TrainController* machine for the European Train Control System.

of events is the conjunction of all the guards of the members of the collection (and not their individual guards as written). We make provision for this by formally scheduling 'event clusters' rather than events, where an event cluster is either a synchronised set of events, or an individual unsynchronised event. Note that event clusters need not necessarily be specified by the synchronisation mechanisms we described in previous sections —it is sufficient that they consist of either all mode events or all pliant events— making specification of event clusters orthogonal to their semantics.

The second is the fact that different machines can undergo mode transitions at different times. This means that unrelated machines are pursuing pliant transitions at those moments. Since the mathematics that guarantees existence of the pliant behaviours is global in nature, a mode transition in machine $A$ must be handled using an (essentially seamless) 'interrupt and resume' construction for any machine $B$ that is pursuing a pliant transition just then. We need to take care that this does not introduce undesired choice points for the runtime scheduler.

The above remarks make it clear that these issues belong predominantly to bureaucracy. Although they add some complexity to the semantics, they add no profundity, so the underlying ideas for multi-machine Hybrid Event-B remain as for single machines.

We turn to the semantics itself. In order to not waste space repeating routine material, we rely on [2] (especially Chapters 5, 9, 14) for the semantics of discrete Event-B; and on [22] (especially Chapter III

§10) for differential equations in the sense of Carathéodory.

For simplicity, our semantics will do a fair bit of 'runtime checking'. In a practical system, most of this is avoided by imposing syntactic tests, and these would normally be provably sound against the semantics (see Section 12). For our purposes though, avoiding dealing with both syntactic and semantic checking simplifies the presentation of the semantics.

Firstly, we make precise a few points of terminology and convention.

- Time, referred to as $t$, takes values in the real left-closed right-open set $[t_0 \ldots +\infty)$, where $t_0$ is an inital value for time. For every other system variable *var*, there is a type $U^{var}$. If *var* is pliant, then $U^{var}$ is $\mathbb{R}$. (1)
- The semantics is given for a project $\mathcal{P}$, which is a set of machines and interfaces, $\mathcal{P} \equiv \{M_1 \ldots M_k, I_1 \ldots I_p\}$. For each state variable *var*, there is exactly one machine or interface that declares it. (2)
- Time is a distinguished variable (read-only, never assigned by events, and synchronised with all machines in $\mathcal{P}$ during *INITIALISATION*). All state variables have interpretations which are functions of an interval of time starting at $t_0$; see (6). As well as directly referring to the time variable, time may be handled indirectly by using clock variables (declared as such), whose values may be reset by mode events. (3)
- The events $M.Ev$ of a machine $M \in \mathcal{P}$ consist of mode events and pliant events. Given a project $\mathcal{P}$, an event cluster is a non-empty set of events, either all mode events (a mode event cluster), or all pliant events (a pliant event cluster). N.B. An event cluster has no internal communications, though there may be bound variables shared across several events of the cluster. (4)
- Given a mode event cluster, given a valuation of all the state variables, and values of inputs and local parameters of the events in the cluster, and a value of time, the mode event cluster is *enabled* iff the tuple of values lies in the topological closure of the set of tuples of values such that all the WHERE clauses of the events in the cluster evaluate to true. Given a pliant event cluster, given a valuation of all the state variables, and a value of time, the pliant event cluster is *enabled* iff all the INIT and WHERE clauses evaluate to true. (5)
- The semantics of $\mathcal{P}$ is a set of system traces $\mathcal{S}$. Each system trace $S \in \mathcal{S}$ is given by a time interval $\mathcal{T} = [t_0 \ldots t_{\text{FINAL}})$ (where $t_{\text{FINAL}}$, with $t_{\text{FINAL}} > t_0$, is finite or $+\infty$), and a set of time dependent variable interpretations $\zeta_{var} : \mathcal{T} \to U^{var}$, one for each state variable *var* of $\mathcal{P}$. If $\mathcal{S}$ is empty we say that the semantics of $\mathcal{P}$ is VOID. (N.B. For reasons of simplicity, we omit inputs, local parameters and outputs from system traces. These are regarded as existing only for the duration of the transitions that they belong to; i.e., the single time value at which a mode transition occurs, or the topological interior of the interval during which a pliant transition takes place. With additional machinery, such parameters could be included in system traces.) (6)
- A pliant variable is *governed* by a pliant event iff, it is constrained in the COMPLY clause of the event, or it occurs in the left hand side of a differential equation or of a direct assignment in the SOLVE clause of the event. To manage pliant variables over the mode transitions of other machines, additional machinery is needed. In a given system trace $S$, for each pliant variable *pli* and time $t \in [t_0 \ldots t_{\text{FINAL}})$, the function $MchPliEv(pli,t)$ returns $M.PliEv$, which consists of a pliant event $PliEv$ and the machine $M \in \mathcal{P}$ that declares $PliEv$ — and is such that the interpretation $\zeta_{pli}$ at time $t$ of variable *pli* is recording the trajectory of a pliant transition determined by the pliant event $M.PliEv$. The function $MchPliEv$ (like the system trace $S$) is constructed incrementally. If the construction breaks down at some point, then the trace $S$ is aborted. (7)

- The set of traces $\mathcal{S}$ is constructed by the step by step process below, which describes how individual system traces are constructed incrementally.[9] Whenever a CHOOSE is encountered, the current trace-so-far is replicated as many times as there are different possible choices, a different choice is allocated to each copy, and the procedure is continued for each resulting trace-so-far. Whenever a TERMINATE is encountered, the current trace-so-far is complete and is added to the semantics $\mathcal{S}$, of $\mathcal{P}$. Whenever an ABORT is encountered, the current trace-so-far is abandoned (and eliminated from $\mathcal{S}$). If a VOID is encountered, or the constructed $\mathcal{S}$ is empty, the semantics is VOID. (8)

The construction of system traces is as follows.

**[1]** Let $\eta := 0$ (where $\eta$ is a meta-level variable).

**[2]** Assuming all machines and interfaces in $\mathcal{P}$ have feasible *INITIALISATION*s that satisfy all the invariants in $\mathcal{P}$, CHOOSE such an initial assignment, thereby interpreting the values of all variables at time $t_0$. Otherwise, VOID.

**[3]** If any non-*INITIALISATION* mode event cluster of $\mathcal{P}$ that does not have any inputs (but which may have local parameters or outputs), is enabled when the state variables have the values at $t_\eta$ and enabling values exist for the local variables, then ABORT.

**[4]** With the state variables having the values at $t_\eta$, CHOOSE a maximal set *PliEvENclst* of *EN*abled pliant event clusters, such that for each machine $M \in \mathcal{P}$ there is at most one event cluster in *PliEvENclst* containing an event of $M$.

  **[4.1]** Considering all occurrences of differential equations and direct assignments in all SOLVE clauses of the events in the event clusters in *PliEvENclst*, if any pliant variable *pli* appears in the left hand side of more than one occurrence, or *pli* is governed by more than one event, then ABORT.

**[5]** If $\eta = 0$ let $PliEvCTclst := \emptyset$. Otherwise, let *PliEvCTclst* be a set of (*C*on*T*inuing) pliant event clusters from machines of $\mathcal{P}$, such that: (1) *PliEvCTclst* is maximal; (2) no variable governed by any event in any event cluster in *PliEvCTclst* is governed by any event in any event cluster in *PliEvENclst*; (3) for every variable *pli* governed by an event *PliEv* in an event cluster in *PliEvCTclst*, we have $\overrightarrow{MchPLiEv(pli, t_\eta)} = M.PliEv$, where $M$ is the machine that contains *PliEv* (and the overarrow denotes left-limit); (4) for every event $PliEv_{CT}$ in any event cluster of *PliEvCTclst*, for every mode variable *mv* which occurs in the WHERE guard of $PliEv_{CT}$, $\overrightarrow{mv(t_\eta)} = mv(t_\eta)$.

**[6]** Let *PliREM* consist of any remaining pliant variables that are not governed by any event in any event cluster in $PliEvENclst \cup PliEvCTclst$. If *PliREM* is nonempty, then ABORT. If there is a machine $M$ such that none of the events in the event clusters in $PliEvENclst \cup PliEvCTclst$ are declared in $M$, then ABORT.

**[7]** ABORT if there does *not* exist a $t_{\mathrm{MAX}} > t_\eta$ such that in the left-closed, right-open interval $[t_\eta \ldots t_{\mathrm{MAX}})$: (1) there is a simultaneous solution of all the differential equations and direct assignments in the SOLVE clauses (that also respects all the *BDApred* predicates) of all the pliant events in all the event clusters in $PliEvENclst \cup PliEvCTclst$, such that; (2) intial state variable values are the values at $t_\eta$, intial input and local parameter values are the right-limit values at $t_\eta$, and such that; (3) the initial values satisfy the INIT and WHERE guards of the pliant events in *PliEvENclst* but need only satisfy the WHERE guards of the pliant events in *PliEvCTclst*.

[9]N.B. The process is not intended to be an executable sequential procedure. All traces-so-far are intended to be explored simultaneously and to completion, even if completion involves an infinite amount of time for a non-terminating system trace.

**[8]** Otherwise, CHOOSE a simultaneous solution as in **[7]**, let $t_{\mathrm{MAX}}$ be maximal such that the properties in **[7]** hold, and use the solution to assign the values of all pliant variables (and outputs) in the interval $[t_\eta \ldots t_{\mathrm{MAX}})$.

**[8.1]** For all pliant variables *pli*, for all $t \in [t_\eta \ldots t_{\mathrm{MAX}})$, let $MchPliEv(pli,t)$ be the event and machine governing the behaviour of *pli*. (N. B. This assignment is total by **[6]**, and is unambiguous by **[4.1]** and **[5]**.)

**[8.2]** For every mode variable, extend its value at $t_\eta$ to a constant function in the interval $[t_\eta \ldots t_{\mathrm{MAX}})$.

**[9]** If no non-*INITIALISATION* mode event cluster is enabled by the values of the state variables at any time $t_{\mathrm{NEXT}}$ in the open interval $(t_\eta \ldots t_{\mathrm{MAX}})$ (including left-limit at $t_{\mathrm{MAX}}$ itself), together with a choice of values for inputs and local parameters, then if the invariants in the machines and interfaces of $\mathcal{P}$ are not satisfied in the open interval $(t_\eta \ldots t_{\mathrm{MAX}})$, then ABORT. Otherwise TERMINATE.

**[10]** CHOOSE $t_{\eta+1} > t_\eta$ such that: **either** $t_{\eta+1}$ is the earliest time at which a non-*INITIALISATION* mode event cluster without input from the environment in any of its events (but potentially having suitably chosen local parameters) is enabled according to the criteria in **[9]** at $t_{\eta+1}$, **or** a non-*INITIALISATION* mode event cluster with at least one event having input from the environment is enabled (with a suitable choice of inputs and local parameters) according to the criteria in **[9]** at $t_{\eta+1}$ and there is no non-*INITIALISATION* mode event cluster without input from the environment that is enabled according to those criteria at any time between $t_\eta$ and $t_{\eta+1}$.

**[11]** If the invariants in the machines and interfaces of $\mathcal{P}$ are not satisfied in the open interval $(t_\eta \ldots t_{\eta+1})$, then ABORT.

**[12]** Let $\eta := \eta + 1$.

**[12.1]** With the state variables having the values at $t_\eta$ (or their left-limit values at $t_\eta$ if $t_\eta = t_{\mathrm{MAX}}$), CHOOSE *MoEvENclst* to be a non-empty set of non-*INITIALISATION* mode event clusters comprising the union of: (a) a maximal (but possibly empty) set of mode event clusters none of whose events has input from the environment, (b) a (possibly empty) set of mode event clusters, each containing at least one event with input from the environment; such that all events are enabled when suitable values are assigned to local parameters, and for each machine $M \in \mathcal{P}$ there is at most one event cluster in *MoEvENclst* containing an event of $M$.

**[12.2]** CHOOSE an assignment to state variables and outputs according to the *BApred* predicates of all the mode events in *MoEvENclst*, such that all the invariants of $\mathcal{P}$ are satisfied, thereby (re)interpreting those variable values at time $t_\eta$. Otherwise ABORT.

**[12.3]** For any other state variable *var* without a value at $t_\eta$, interpret its value at $t_\eta$ as $\overrightarrow{var(t_\eta)}$, provided this is finite. Otherwise ABORT.

**[12.4]** Discard the interpretation of all state variables in the open interval $(t_\eta \ldots t_{\mathrm{MAX}})$, where $t_{\mathrm{MAX}}$ is the value determined in **[8]**. (If $t_\eta = t_{\mathrm{MAX}}$ then the interval is empty.)

**[13]** Goto **[3]**.

Regarding the soundness of the above construction, we make the following remarks. In PaperI, we discussed the soundness of the semantic construction for single machines based on the assumed soundness of update semantics for discrete Event-B, and on the soundness of the Carathéodory semantics for DEs and the demand for piecewise absolute continuity of pliant behaviours. In this paper, we aim to bootstrap the single machine definition to a multi-machine definition, by arguing that the occurrences of mode events and pliant events in the multi-machine context can be interpreted as occurrences of mode and pliant events in a (fictitious) global context, built out of the ingredients of the individual machines and interfaces of the multi-machine project.

More precisely, suppose we have a multi-machine project $\mathcal{P}$. We construct a single machine $\mathcal{M}$ out of the ingredients of $\mathcal{P}$. This is predominantly a lexical job, undertaken within the constraints of the name space conventions of Section 3.1. We then argue that the system traces of $\mathcal{P}$ can be identified with a subset of the system traces of $\mathcal{M}$. Since $\mathcal{M}$ is consistent by PaperI, consistency of the multi-machine construction follows.

We assume that the various alphabets of names are countably infinite, and that we have at our disposal a renaming relation "_" which takes its argument (which is typically some data structure manufactured from lexical elements of $\mathcal{P}$) and maps it injectively to a fresh name in the relevant name space. Since the name spaces are countably infinite and $\mathcal{P}$ has a finite description, we can safely assume that "_" is always able to return a suitable value.

We will assume that none of the static CONTEXTs SEEn by any part of $\mathcal{P}$ contains any identifier definition (of some static piece of mathematics) that conflicts with any other such identifier definition. Therefore we can take the scope of all the definitions in all such CONTEXTs to be project-wide for convenience.

We construct $\mathcal{M}$ as follows. We work exclusively with the *instantiated* project $\mathcal{P}^I$, that is to say, the project description that results when all the MACHINE ... IS ... clauses have been evaluated.

Since the scope of state variable names in $\mathcal{P}^I$ is global, we can take the state variables of $\mathcal{M}$ to just be the variables of $\mathcal{P}^I$, verbatim. Because of this, we can then immediately take the invariants of $\mathcal{M}$ to just be the invariants of all the machines and interfaces of $\mathcal{P}^I$. And a similar approach will work for any syntactic element of any machine or interface in $\mathcal{P}^I$ defined in terms of state variables alone. For example, any VARIANT expression of any machine of $\mathcal{P}^I$ becomes a variant of $\mathcal{M}$. And since CONTEXTs are static, consistent, and do not require renaming, any INITIALISATION of an interface or machine of $\mathcal{P}^I$ becomes an initialisation of $\mathcal{M}$.

Next we define the event names of $\mathcal{M}$. Let $M_1.Ev_1, M_1.Ev_2, \ldots, M_p.Ev_q$ be a listing (in some canonical, say lexicographical, ordering) of the events in a nonempty set *EvClst* of event clusters, where either: each element of *EvClst* is a mode event cluster, or, each element of *EvClst* is a pliant event cluster; and such that for each machine $M$ of $\mathcal{P}^I$, at most one element of *EvClst* contains an event of $M$. Construct the event name "$M_1.Ev_1, M_1.Ev_2, \ldots, M_p.Ev_q$". If any of $M_1.Ev_1, M_1.Ev_2, \ldots, M_p.Ev_q$ have a parameter —say $M_1.Ev_2$ has input parameter $M_1.Ev_2.in?$— we create a corresponding input parameter "$M_1.Ev_2.in$"? for "$M_1.Ev_1, M_1.Ev_2, \ldots, M_p.Ev_q$". Any guard of "$M_1.Ev_1, M_1.Ev_2, \ldots, M_p.Ev_q$", whether an ANY guard or an INIT guard, is created as the conjunction of the corresponding guards of $M_1.Ev_1, M_1.Ev_2, \ldots, M_p.Ev_q$, with occurrences of parameter names replaced suitably. Similarly for COMPLY clauses. The actions of "$M_1.Ev_1, M_1.Ev_2, \ldots, M_p.Ev_q$", whether mode actions or pliant actions, consist of just the union of the sets of actions of the $M_1.Ev_1, M_1.Ev_2, \ldots, M_p.Ev_q$, with occurrences of parameter names replaced suitably.

Having constructed the event named "$M_1.Ev_1, M_1.Ev_2, \ldots, M_p.Ev_q$", if *EvClst* consisted of clusters of pliant events, we next do the following in all possible ways. Regarding an overline as additional lexical element, we construct the event name "$M_1.Ev_1, \overline{M_1.Ev_2}, \ldots, \overline{M_p.Ev_q}$" where the overlined event names are *all* events belonging to *some (but not all)* of the event clusters in *EvClst*. If *EvClst* contains only one cluster this construction is null. For each event name thus constructed, we give it a body identical to that of "$M_1.Ev_1, M_1.Ev_2, \ldots, M_p.Ev_q$" except that the INIT guards of the overlined contributing events are removed (i.e. set to true).

We will assume that "$M_1.INITIALISATION, M_1.INITIALISATION, \ldots, M_p.INITIALISATION$" is the *INITIALISATION* of $\mathcal{M}$, where the list of initialisations includes all the initialisations of all the machines and interfaces of $\mathcal{P}^I$, thus completing the construction of the events of $\mathcal{M}$.

Finally, we let $\mathcal{M}$ SEE all the CONTEXTs seen by any of the constituents of $\mathcal{P}^I$. Since we have assumed CONTEXTs do not contain inconsistent declarations, and do not mention state variables, they are not subject to renaming, and so may be taken verbatim.

**Definition 11.1.** *Let $\mathcal{P}$ be a project, $\mathcal{P}^I$ be the instantiated project, and $\mathcal{M}$ the machine constructed above. $\mathcal{M}$ is called an associated single machine of $\mathcal{P}$ (and of $\mathcal{P}^I$).*

We say *an* associated single machine, not *the* associated single machine, since we do not regard "_" as a fixed global static relation, but rather, as a sufficiently injective relation constructed *ad hoc* per project.

**Theorem 11.2.** *Let $\mathcal{P}$ be a project, $\mathcal{P}^I$ be its instantiation, and $\mathcal{M}$ an associated single machine. Up to the renaming defined by* "_"*, every system trace of $\mathcal{P}$ is a system trace of $\mathcal{M}$.*

*Proof:* Because we only evaluated "_" on fully hierarchical arguments (i.e. arguments that —according to the name space conventions defined in Section 3.1— contained sufficient data to disambiguate potential name clashes arising from name coincidences in local name spaces), the assumed sufficient injectivity of "_" ensures no inadvertent clashes occur in $\mathcal{M}$.[10]

It remains to check that every system trace of $\mathcal{P}$ corresponds to a system trace of $\mathcal{M}$. We proceed by induction. Evidently the initial states of $\mathcal{P}$ and $\mathcal{M}$ are the same up to the renaming of the initialisations. For the inductive step, we assume the states of $\mathcal{P}$ and $\mathcal{M}$ are the same after $N$ steps. Then there are two cases: for mode event steps and for pliant event steps.

Suppose $\mathcal{P}$ schedules a number of mode event clusters in step **[12]** of the semantics, consisting of events, say $M_1.Ev_1, M_1.Ev_2, \ldots, M_p.Ev_q$. Then it is clear that the event "$M_1.Ev_1, M_1.Ev_2, \ldots, M_p.Ev_q$" of $\mathcal{M}$ is similarly schedulable. And since the states of $\mathcal{P}$ and $\mathcal{M}$ are the same, the same state change is possible in both $\mathcal{P}$ and $\mathcal{M}$, completing the inductive step.

Suppose $\mathcal{P}$ schedules a number of pliant event clusters in step **[8]** of the semantics, consisting of events, say $M_1.Ev_1, M_1.Ev_2, \ldots, M_p.Ev_q$, where some of these events belong to *PliEvENclst* and some to *PliEvCTclst*. Then it is clear that the event "$M_1.Ev_1, \overline{M_1.Ev_2}, \ldots, \overline{M_p.Ev_q}$" of $\mathcal{M}$ (where the overlined event names correspond exactly to the events in the clusters in *PliEvCTclst*), is similarly schedulable. And since the states of $\mathcal{P}$ and $\mathcal{M}$ are the same, the same state change is possible in both $\mathcal{P}$ and $\mathcal{M}$, completing the inductive step. □

Note that we do not attempt to prove an equivalence between the semantics of $\mathcal{P}$ and $\mathcal{M}$. The scheduling policy of $\mathcal{M}$ (i.e. *any* enabled event can run) is too liberal to restrict the behaviour to just that of $\mathcal{P}$. To get an equivalence, we would have to introduce suitable *priorities* into the scheduling policy in order to prevent undesirable scheduling choices.

**Definition 11.3.** *A Hybrid Event-B project $\mathcal{P}$ is said to be* non-void *iff its semantics is not* VOID, *i.e. its set of system traces $\mathcal{S} \neq \emptyset$. It is said to be* correct *iff it is* non-void, *and also, during the construction of its semantics, no* ABORT *is ever encountered.*

**Theorem 11.4.** *Let $\mathcal{P}$ be a project, $\mathcal{P}^I$ be its instantiation, and $\mathcal{M}$ an associated single machine. Then $\mathcal{P}$ is* correct *if $\mathcal{M}$ is* correct *(in the single machine sense of Definition 7.1 of PaperI).*[11]

*Proof:* Evident, from the fact that if no system trace construction attempt of $\mathcal{M}$ ever encounters an ABORT, then the subset of those attempts corresponding to $\mathcal{P}$ also doesn't. □

**Definition 11.5.** *A Hybrid Event-B project $\mathcal{P}R$ correctly refines a Hybrid Event-B project $\mathcal{P}$ iff for every system trace SR of $\mathcal{P}R$ there is a system trace S of $\mathcal{P}$ such that:*

*(i) If SR occupies the time interval $[t_0 \ldots t_{\mathrm{FIN}R})$, then S occupies a time interval $[t_0 \ldots t_{\mathrm{FIN}})$, where $t_{\mathrm{FIN}R} \leq t_{\mathrm{FIN}}$.*
*(ii) For each t in $[t_0 \ldots t_{\mathrm{FIN}R})$, all the invariants hold.*
*(iii) At each occurrence of a mode event in S there is an occurrence of a mode event in SR.*

[10]E.g. even if $Ev$ is an event name in both machine $M_1$ and machine $M_2$, $M_1.Ev$ and $M_2.Ev$ do not clash, and thus neither do "$M_1.Ev$" and "$M_2.Ev$".

[11]The single machine Definition 7.1 of PaperI is like Definition 11.3, with the substitution of 'machine' for 'project'.

A brief inspection shows that Definition 11.5 of a correct refinement for projects is identical to the definition of a correct refinement for single machines from PaperI, with the obvious substitution of 'project' for 'machine'. This is made possible by the fact that in both cases it is the set of system traces that is the key concept.

## 12. Correctness

In this section we consider how the correctness of a multi-machine system can be safely established by discharging a collection of statically derivable proof obligations (POs) — the objective of these is to enable us to conclude statically, that runtime errors do not occur. In addressing this goal there is always a tension between the strength of the assumptions made, and the effort expended in proving the relevant correctness properties. We start by summarising the single machine situation.

### *12.1. Single Machine POs, and Associated Single Machines*

In Figs. 19-21, we collect the POs relevant to a single Hybrid Event-B machine and its refinement that were established in PaperI (where full details can be found). We briefly review them now.

Fig. 19 has the POs that establish soundness of an individual machine. Init/FIS (PaperI (11)) proves the feasibility of initialisation; Init/INV (PaperI (12)) shows that initialisation establishes the invariants. MoEv/FIS (PaperI (13)) is traditional mode event feasibility, i.e. there is an after-state reachable from the before-state. PliEv/FIS (PaperI (14)) is pliant event feasibility, i.e. there is a time-indexed family of after-states given by *SOL*, satisfying the SOLVE clause and the *BDApred* clause. Optionally (heavy **[ ]** brackets) it asks that the duration of the solution is greater than a Zeno constant. MoEv/INV (PaperI (15)) is traditional mode event invariant preservation. PliEv/INV (PaperI (16)) is invariant preservation for pliant events, demanding that this applies until the preemption time defined using TRM, this in turn being calculated using the disjunction of mode event guards.[12] MoPli/WFor, well-formedness (PaperI (17)), controls the handover from mode to pliant events: if a mode event makes a valid step, it disables all mode events and enables some pliant event. PliMo/WFor (PaperI (18)) does the converse: if a pliant event runs for a MAXIMAL time during which no mode event is enabled, then (assuming it is not a FINAL event), the termination time must be WELLDEFined and the state values then must enable a mode event.

Figs. 20 and 21 collect the POs that confirm soundness of a refinement. Init/FISR (PaperI (20)) is feasibility of refined intialisation, while Init/INVR (PaperI (21)) demands that initalisation establishes the joint invariant *K* (for a suitable abstract counterpart value). MoEv/FISR (PaperI (22)) checks feasibility of a refining mode event, in the presence of a suitable abstract counterpart value. MoEv/GRDR (PaperI (23)) is mode guard strengthening. MoEv/INVR (PaperI (24)) is conventional simulation for mode events. The next three POs, MoEv/FISRW, MoEv/GRDRW and MoEv/INVRW (PaperI (25)-(27)), redo the work of the previous three —with the help of a witness relation *W* that is able to supply the existential witnesses required by the original versions— provided the witness relation *W* itself is feasible (MoEv/FISRW). MoEv/NewR (PaperI (28)) is the simplified invariant preservation PO when new refining mode events refine 'skip', and MoEv/NewRV (PaperI (29)) demands that such new mode events decrease a VARIANT function *V*. MoEv/RelDLF (PaperI (35)) is mode event relative deadlock freedom: if one or more of the abstract mode events is enabled, then at least one refining mode event is enabled.

Then come the pliant events. PliEv/FISR (PaperI (30)) is feasibility of refining pliant events, in the presence of suitable abstract corresponding values. PliEv/GRDR (PaperI (31)) is pliant guard strengthening, including the optional removal of checking abstract INIT guards (the *PliEvCTclst* case). PliEv/INVR (PaperI (32)) is the pliant simulation condition, ensuring a time-parameterised family of simulations at

[12]N. B. TRM is omitted in the case of FINAL pliant events.

Init/FIS $\exists u' \bullet \mathit{Init}_A(u')$

Init/INV $\mathit{Init}_A(u') \Rightarrow I(u')$

MoEv/FIS $I(u) \wedge \mathit{grd}_{\mathit{MoEvA}}(u,i?,l) \Rightarrow (\exists u', o! \bullet \mathit{BApred}_{\mathit{MoEvA}}(u,i?,l,o!,u'))$

PliEv/FIS
$$\begin{aligned}&I(u(\mathbb{t}_{\mathrm{L}})) \wedge \mathit{iv}_{\mathit{PliEvA}}(u(\mathbb{t}_{\mathrm{L}})) \wedge \mathit{grd}_{\mathit{PliEvA}}(u(\mathbb{t}_{\mathrm{L}}))\\ &\quad \Rightarrow (\exists \mathbb{t}_{\mathrm{R}} > \mathbb{t}_{\mathrm{L}} \bullet [\ (\mathbb{t}_{\mathrm{R}} - \mathbb{t}_{\mathrm{L}} \geq \delta_{\mathsf{Zeno}\mathit{PliEvA}}) \wedge\ ]\ (\forall \mathbb{t}_{\mathrm{L}} \leq t < \mathbb{t}_{\mathrm{R}} \bullet (\exists u(t), i?(t), l(t), o!(t) \bullet\\ &\qquad \mathit{BDApred}_{\mathit{PliEvA}}(u(t),i?(t),l(t),o!(t),t) \wedge \mathit{SOL}_{\mathit{PliEvA}}(u(t),i?(t),l(t),o!(t),t))))\end{aligned}$$

MoEv/INV $I(u) \wedge \mathit{grd}_{\mathit{MoEvA}}(u,i?,l) \wedge \mathit{BApred}_{\mathit{MoEvA}}(u,i?,l,o!,u') \Rightarrow I(u')$

PliEv/INV
$$\begin{aligned}&I(u(\mathbb{t}_{\mathrm{L}})) \wedge \mathit{iv}_{\mathit{PliEvA}}(u(\mathbb{t}_{\mathrm{L}})) \wedge \mathit{grd}_{\mathit{PliEvA}}(u(\mathbb{t}_{\mathrm{L}})) \wedge (\exists \mathbb{t}_{\mathrm{R}} > \mathbb{t}_{\mathrm{L}} \bullet \mathrm{TRM}(\mathbb{t}_{\mathrm{R}}) \wedge (\forall \mathbb{t}_{\mathrm{L}} \leq t < \mathbb{t}_{\mathrm{R}},\\ &u(t),i?(t),l(t),o!(t) \bullet \mathit{BDApred}_{\mathit{PliEvA}}(u(t),i?(t),l(t),o!(t),t) \wedge \mathit{SOL}_{\mathit{PliEvA}}(u(t),i?(t),l(t),o!(t),t)))\\ &\quad \Rightarrow (\forall \mathbb{t}_{\mathrm{L}} \leq t < \mathbb{t}_{\mathrm{R}} \bullet I(u(t)))\end{aligned}$$

MoPli/WFor
$$\begin{aligned}&\exists u_0, i_0?, l_0, o_0! \bullet I(u_0) \wedge \mathit{grd}_{\mathit{MoEvA}}(u_0,i_0?,l_0) \wedge \mathit{BApred}_{\mathit{MoEvA}}(u_0,i_0?,l_0,o_0!,u) \wedge I(u)\\ &\quad \Rightarrow \neg\,[\ \exists l \bullet \mathit{grd}_{\mathit{MoEvA1}}(u,l) \vee \mathit{grd}_{\mathit{MoEvA2}}(u,l) \ldots \mathit{grd}_{\mathit{MoEvAN}}(u,l)\ ] \wedge\\ &\qquad [\ (\mathit{iv}_{\mathit{PliEvA1}}(u) \wedge \mathit{grd}_{\mathit{PliEvA1}}(u)) \vee (\mathit{iv}_{\mathit{PliEvA2}}(u) \wedge \mathit{grd}_{\mathit{PliEvA2}}(u)) \vee \ldots \vee\\ &\qquad (\mathit{iv}_{\mathit{PliEvAM}}(u) \wedge \mathit{grd}_{\mathit{PliEvAM}}(u))\ ]\end{aligned}$$

PliMo/WFor
$$\begin{aligned}&I(u(\mathbb{t}_{\mathrm{L}})) \wedge \mathit{iv}_{\mathit{PliEvA}}(u(\mathbb{t}_{\mathrm{L}})) \wedge \mathit{grd}_{\mathit{PliEvA}}(u(\mathbb{t}_{\mathrm{L}})) \wedge (\exists \mathbb{t}_{\mathrm{R}} > \mathbb{t}_{\mathrm{L}} \bullet (\forall \mathbb{t}_{\mathrm{L}} \leq t < \mathbb{t}_{\mathrm{R}}, u(t), i?(t), l(t), o!(t) \bullet\\ &\mathit{BDApred}_{\mathit{PliEvA}}(u(t),i?(t),l(t),o!(t),t) \wedge \mathit{SOL}_{\mathit{PliEvA}}(u(t),i?(t),l(t),o!(t),t) \wedge \mathsf{MAXIMAL}(\mathbb{t}_{\mathrm{R}}) \wedge\\ &\neg\,[\ \exists i?, l \bullet \mathit{grd}_{\mathit{MoEvA1}}(u(t),i?,l) \vee \mathit{grd}_{\mathit{MoEvA2}}(u(t),i?,l) \vee \ldots \vee \mathit{grd}_{\mathit{MoEvAN}}(u(t),i?,l)\ ]))\\ &\quad \Rightarrow \mathsf{WELLDEF}(\mathbb{t}_{\mathrm{R}}) \wedge [\ \exists i?, l \bullet \mathit{grd}_{\mathit{MoEvA1}}(\overrightarrow{u(\mathbb{t}_{\mathrm{R}})}, i?, l) \vee \mathit{grd}_{\mathit{MoEvA2}}(\overrightarrow{u(\mathbb{t}_{\mathrm{R}})}, i?, l) \vee \ldots \vee\\ &\qquad\qquad \mathit{grd}_{\mathit{MoEvAN}}(\overrightarrow{u(\mathbb{t}_{\mathrm{R}})}, i?, l)\ ]\end{aligned}$$

Figure 19: Single machine soundness POs from PaperI.

individual moments of time. PliEv/FISRW (PaperI (33)) and PliEv/INVRW (PaperI (34)) do the same job as PliEv/FISR and PliEv/INVR, but with the help of a time-dependent witness relation $W$. Note that there is no PliEv/GRDRW, since the guards of pliant events cannot depend on inputs (so that there is no existential quantification in PliEv/GRDR). Finally there is PliEv/RelDLF (PaperI (36)), essentially, the same as its relative deadlock freedom mode counterpart.

Theorem 9.1 of PaperI assures us that a single machine is correct if it satisfies the POs of Fig. 19. Accordingly we can conclude:

**Theorem 12.1.** *If an associated single machine $\mathcal{M}$ of a multi-machine project $\mathcal{P}$ satisfies the POs of Fig. 19 (interpreted as concerning $\mathcal{M}$), then $\mathcal{P}$ is correct in the sense of Definition 11.3.*

*Proof:* Since the POs guarantee correctness of any single machine (including the associated single machine of the project), and the associated single machine's system traces are a superset of the system traces of the multi-machine project, the latter is correct too. ☐

**Theorem 12.2.** *Suppose project $\mathcal{PR}$ is a syntactic refinement of project $\mathcal{P}$ — in the sense that it is connected syntactically to $\mathcal{P}$ in the way expected for a refinement, and also obeys the restrictions described in Section 5 for refinements of projects. Suppose $\mathcal{MR}$ is an associated single machine of $\mathcal{PR}$ and the refinement POs of Figs. 20 and 21 hold between $\mathcal{M}$ (the associated single machine of $\mathcal{P}$) and $\mathcal{MR}$. Then $\mathcal{PR}$ is a correct refinement of $\mathcal{P}$.*

*Proof:* Provided the POs hold for $\mathcal{M}$ and $\mathcal{MR}$, Theorem 9.2 of PaperI assures us that $\mathcal{MR}$ refines $\mathcal{M}$ in the sense of Definition 8.1 of PaperI. This means that: the abstract system trace's temporal duration is at least as long as the concrete one's; that all the invariants in both systems hold throughout the common

| | |
|---|---|
| Init/FISR | $\exists w' \bullet Init_C(w')$ |
| Init/INVR | $Init_C(w') \Rightarrow (\exists u' \bullet Init_A(u') \wedge K(u',w'))$ |
| MoEv/FISR | $\exists u \bullet K(u,w) \wedge grd_{MoEvC}(w,j?,k) \Rightarrow (\exists w',p! \bullet BApred_{MoEvC}(w,j?,k,p!,w'))$ |
| MoEv/GRDR | $I(u) \wedge K(u,w) \wedge grd_{MoEvC}(w,j?,k) \Rightarrow (\exists i?,l \bullet grd_{MoEvA}(u,i?,l))$ |
| MoEv/INVR | $I(u) \wedge K(u,w) \wedge grd_{MoEvC}(w,j?,k) \wedge BApred_{MoEvC}(w,j?,k,p!,w')$ <br> $\Rightarrow (\exists i?,l,o!,u' \bullet BApred_{MoEvA}(u,i?,l,o!,u') \wedge K(u',w'))$ |
| MoEv/FISRW | $I(u) \wedge K(u,w) \wedge grd_{MoEvC}(w,j?,k) \wedge BApred_{MoEvC}(w,j?,k,p!,w')$ <br> $\Rightarrow (\exists i?,l,o!,u' \bullet W(u,i?,l,o!,u',w,j?,k,p!,w'))$ |
| MoEv/GRDRW | $I(u) \wedge K(u,w) \wedge grd_{MoEvC}(w,j?,k) \wedge W(u,i?,l,o!,u',w,j?,k,p!,w')$ <br> $\Rightarrow grd_{MoEvA}(u,i?,l)$ |
| MoEv/INVRW | $I(u) \wedge K(u,w) \wedge grd_{MoEvC}(w,j?,k) \wedge BApred_{MoEvC}(w,j?,k,p!,w') \wedge W(u,i?,l,o!,u',w,j?,k,p!,w')$ <br> $\Rightarrow BApred_{MoEvA}(u,i?,l,o!,u') \wedge K(u',w')$ |
| MoEv/NewR | $I(u) \wedge K(u,w) \wedge grd_{NewEvC}(w,j?,k) \wedge BApred_{NewEvC}(w,j?,k,p!,w') \Rightarrow K(u,w')$ |
| MoEv/NewRV | $BApred_{NewEvC}(w,j?,k,p!,w') \Rightarrow V(w') < V(w)$ |
| MoEv/RelDLF | $I(u) \wedge K(u,w) \wedge (\exists o!,p!,u',w' \bullet W(u,i?,l,o!,u',w,j?,k,p!,w')) \wedge$ <br> $[\ grd_{MoEvA1}(u,i?,l) \vee grd_{MoEvA2}(u,i?,l) \vee \ldots \vee grd_{MoEvAN}(u,i?,l)\ ]$ <br> $\Rightarrow grd_{MoEvC1}(w,j?,k) \vee grd_{MoEvC2}(w,j?,k) \vee \ldots \vee grd_{MoEvCM}(w,j?,k)$ |

Figure 20: Single machine refinement POs from PaperI: initialisation and mode events. Subscripts $A$ and $C$ indicate abstract and concrete entities.

part of the durations; and that each abstract mode event occurrence corresponds to a concrete mode event occurrence.

We proceed by induction, with inductive hypothesis stating that:

*Every system trace-so-far $\mathcal{SR}$ of $\mathcal{PR}$ is a refinement of a system trace-so-far $\mathcal{S}$ of $\mathcal{P}$. The durations of $\mathcal{SR}$ and $\mathcal{S}$ are equal; the values of all state variables after $\mathcal{SR}$ and $\mathcal{S}$ satisfy all the invariants; for each abstract mode event occurrence in $\mathcal{S}$ there is a concrete mode event occurrence in $\mathcal{SR}$.*

Being obviously true for the initial states, we assume the hypothesis holds for $\mathcal{SR}$ up to length $N$, and examine extending some $\mathcal{SR}$ by one step. This step extends $\mathcal{SR}$ to some $\mathcal{SR}'$, by a step of $\mathcal{PR}$. This step corresponds in an obvious way to a step of $\mathcal{MR}$. That $\mathcal{MR}$ step is a refinement of an $\mathcal{M}$ step because the single machine refinement POs hold for $\mathcal{M}$ and $\mathcal{MR}$. We must show that the $\mathcal{M}$ step can always be chosen to correspond to a $\mathcal{P}$ step.

Suppose the $\mathcal{PR}$ step is a mode event step. Mode events in $\mathcal{PR}$ and $\mathcal{MR}$ (and their runtime steps) correspond bijectively. The corresponding $\mathcal{MR}$ step either refines an explicit $\mathcal{M}$ mode step or refines a skip in $\mathcal{M}$. In the former case, noting that mode events (and their runtime steps) in $\mathcal{P}$ and $\mathcal{M}$ also correspond bijectively, we conclude that a step of $\mathcal{P}$ exists that re-establishes the inductive hypothesis. In the latter case, there is no explicit $\mathcal{M}$ mode step, and so the same conclusion holds in $\mathcal{P}$, the inductive hypothesis being easy to re-establish.

Otherwise, the $\mathcal{PR}$ step is a pliant event step. It is some combination of pliant event steps (freshly enabled by the preceding mode step of $\mathcal{PR}$), and of 'interrupt and resume' steps, these being distributed among the machines of $\mathcal{PR}$. This combination corresponds to a single step of a single $\mathcal{MR}$ event via the "_" construction, which offers the correct combination as one of the options in the construction. In turn, this step is a refinement of an $\mathcal{M}$ step, because the POs hold.

Because the restrictions described in Section 5 hold, the joint invariants of the $\mathcal{M}$ to $\mathcal{MR}$ refinement decompose into a conjunction of separate joint invariants for each pair of corresponding interfaces or

PliEv/FISR
$$\begin{aligned}
&(\exists u(\mathbb{t}_{\mathrm{L}}) \bullet I(u(\mathbb{t}_{\mathrm{L}})) \land K(u(\mathbb{t}_{\mathrm{L}}), w(\mathbb{t}_{\mathrm{L}})) \land iv_{PliEvC}(w(\mathbb{t}_{\mathrm{L}})) \land grd_{PliEvC}(w(\mathbb{t}_{\mathrm{L}}))\\
&\quad \Rightarrow (\exists \mathbb{t}_{\mathrm{R}} > \mathbb{t}_{\mathrm{L}} \bullet [\, (\mathbb{t}_{\mathrm{R}} - \mathbb{t}_{\mathrm{L}} \geq \delta_{\mathsf{Zeno}PliEvC}) \land \,]\, (\forall \mathbb{t}_{\mathrm{L}} < t < \mathbb{t}_{\mathrm{R}} \bullet (\exists w(t), j?(t), k(t), p!(t) \bullet\\
&\qquad\quad BDApred_{PliEvC}(w(t), j?(t), k(t), p!(t), t) \land SOL_{PliEvC}(w(t), j?(t), k(t), p!(t), t)))))
\end{aligned}$$

PliEv/GRDR
$$\begin{aligned}
&I(u(\mathbb{t}_{\mathrm{L}})) \land K(u(\mathbb{t}_{\mathrm{L}}), w(\mathbb{t}_{\mathrm{L}})) \land iv_{PliEvC}(w(\mathbb{t}_{\mathrm{L}})) \land grd_{PliEvC}(w(\mathbb{t}_{\mathrm{L}}))\\
&\quad \Rightarrow [\, iv_{PliEvA}(u(\mathbb{t}_{\mathrm{L}})) \land \,]\; grd_{PliEvA}(u(\mathbb{t}_{\mathrm{L}}))
\end{aligned}$$

PliEv/INVR
$$\begin{aligned}
&I(u(\mathbb{t}_{\mathrm{L}})) \land K(u(\mathbb{t}_{\mathrm{L}}), w(\mathbb{t}_{\mathrm{L}})) \land iv_{PliEvC}(w(\mathbb{t}_{\mathrm{L}})) \land grd_{PliEvC}(w(\mathbb{t}_{\mathrm{L}})) \Rightarrow\\
&\quad \big(\exists \mathbb{t}_{\mathrm{R}} > \mathbb{t}_{\mathrm{L}} \bullet \mathrm{TRM}(\mathbb{t}_{\mathrm{R}}) \land (\forall \mathbb{t}_{\mathrm{L}} < t < \mathbb{t}_{\mathrm{R}}, w(t), j?(t), k(t), p!(t) \bullet\\
&\quad BDApred_{PliEvC}(w(t), j?(t), k(t), p!(t), t) \land SOL_{PliEvC}(w(t), j?(t), k(t), p!(t), t))\\
&\qquad \Rightarrow (\forall \mathbb{t}_{\mathrm{L}} < t < \mathbb{t}_{\mathrm{R}} \bullet (\exists u(t), i?(t), l(t), o!(t) \bullet\\
&\qquad\qquad BDApred_{PliEvA}(u(t), i?(t), l(t), o!(t), t) \land SOL_{PliEvA}(u(t), i?(t), l(t), o!(t), t) \land\\
&\qquad\qquad K(u(t), w(t))))\big)
\end{aligned}$$

PliEv/FISRW
$$\begin{aligned}
&I(u(\mathbb{t}_{\mathrm{L}})) \land K(u(\mathbb{t}_{\mathrm{L}}), w(\mathbb{t}_{\mathrm{L}})) \land iv_{PliEvC}(w(\mathbb{t}_{\mathrm{L}})) \land grd_{PliEvC}(w(\mathbb{t}_{\mathrm{L}})) \Rightarrow\\
&\quad \big(\exists \mathbb{t}_{\mathrm{R}} > \mathbb{t}_{\mathrm{L}} \bullet \mathrm{TRM}(\mathbb{t}_{\mathrm{R}}) \land (\forall \mathbb{t}_{\mathrm{L}} < t < \mathbb{t}_{\mathrm{R}}, w(t), j?(t), k(t), p!(t) \bullet\\
&\quad BDApred_{PliEvC}(w(t), j?(t), k(t), p!(t), t) \land SOL_{PliEvC}(w(t), j?(t), k(t), p!(t), t))\\
&\qquad \Rightarrow (\forall \mathbb{t}_{\mathrm{L}} < t < \mathbb{t}_{\mathrm{R}} \bullet (\exists u(t), i?(t), l(t), o!(t) \bullet W(u(t), i?(t), l(t), o!(t), w(t), j?(t), k(t), p!(t)))))\big)
\end{aligned}$$

PliEv/INVRW
$$\begin{aligned}
&I(u(\mathbb{t}_{\mathrm{L}})) \land K(u(\mathbb{t}_{\mathrm{L}}), w(\mathbb{t}_{\mathrm{L}})) \land iv_{PliEvC}(w(\mathbb{t}_{\mathrm{L}})) \land grd_{PliEvC}(w(\mathbb{t}_{\mathrm{L}})) \Rightarrow\\
&\quad \big(\exists \mathbb{t}_{\mathrm{R}} > \mathbb{t}_{\mathrm{L}} \bullet \mathrm{TRM}(\mathbb{t}_{\mathrm{R}}) \land (\forall \mathbb{t}_{\mathrm{L}} < t < \mathbb{t}_{\mathrm{R}}, w(t), j?(t), k(t), p!(t) \bullet\\
&\quad BDApred_{PliEvC}(w(t), j?(t), k(t), p!(t), t) \land SOL_{PliEvC}(w(t), j?(t), k(t), p!(t), t) \land\\
&\quad W(u(t), i?(t), l(t), o!(t), w(t), j?(t), k(t), p!(t)))\\
&\qquad \Rightarrow (\forall \mathbb{t}_{\mathrm{L}} < t < \mathbb{t}_{\mathrm{R}} \bullet\\
&\qquad\quad BDApred_{PliEvA}(u(t), i?(t), l(t), o!(t), t) \land SOL_{PliEvA}(u(t), i?(t), l(t), o!(t), t) \land\\
&\qquad\quad K(u(t), w(t))\big)
\end{aligned}$$

PliEv/RelDLF
$$\begin{aligned}
&I(u) \land K(u(\mathbb{t}_{\mathrm{L}}), w(\mathbb{t}_{\mathrm{L}})) \land [\, (iv_{PliEvA1}(u(\mathbb{t}_{\mathrm{L}})) \land grd_{PliEvA1}(u(\mathbb{t}_{\mathrm{L}}))) \lor\\
&(iv_{PliEvA2}(u(\mathbb{t}_{\mathrm{L}})) \land grd_{PliEvA2}(u(\mathbb{t}_{\mathrm{L}})) \lor \ldots \lor (iv_{PliEvAM}(u(\mathbb{t}_{\mathrm{L}})) \land grd_{PliEvAM}(u(\mathbb{t}_{\mathrm{L}}))\,]\\
&\quad \Rightarrow [\, (iv_{PliEvC1}(w(\mathbb{t}_{\mathrm{L}})) \land grd_{PliEvC1}(w(\mathbb{t}_{\mathrm{L}}))) \lor (iv_{PliEvC2}(w(\mathbb{t}_{\mathrm{L}})) \land grd_{PliEvC2}(w(\mathbb{t}_{\mathrm{L}})) \lor \ldots \lor\\
&\qquad (iv_{PliEvCN}(w(\mathbb{t}_{\mathrm{L}})) \land grd_{PliEvCN}(w(\mathbb{t}_{\mathrm{L}}))\,]
\end{aligned}$$

Figure 21: Single machine refinement POs from PaperI: pliant events.

machines of $\mathcal{P}$ and $\mathcal{PR}$. This implies that the component of the $\mathcal{MR}$ step (whether a freshly enabled pliant step or a resume step) corresponding to each machine, is separately enabled. In turn this implies that each such step is enabled in $\mathcal{P}$, completing this case of the inductive step.

We do the same for each possible way of extending any length $N$ system trace-so-far, completing the inductive step as a whole. □

The above results, while convenient enough conceptually, demand global knowledge, and, in effect, global reasoning. This defeats the object of decomposing a project into a collection of subsystems since the task of verification is made worse by the structuring mechanisms. In the next section, we take measures to reduce the need for global reasoning, by making the structuring mechanisms more precise.

## *12.2. Verification of Multi-Machine Projects*

In this section we extend the single machine formulation of statically derivable correctness to the multi-machine case, without resorting to the single machine reformulation of a multi-machine project. In this context, the tension noted above, between strength of assumptions made and effort expended in proof of soundness based on those assumptions, becomes quite acute. The desire for separate working (for development and verification) pulls powerfully against the desire for the widest global system knowledge (for the most incisive reasoning).

We want to base our statically derivable correctness properties as far as possible on the single machine POs described earlier. Here though, we face a couple of additional difficulties not encountered in the definition of multi-machine runtime semantics.

The first concerns ensuring correct coverage of variable update. In the runtime semantics, if a choice of events scheduled for execution results in inconsistent or multiple assignment to the same variable, or some variable unaccounted for, the run is simply ABORTed. Here, we must prevent such things statically.

The second arises from the presence of synchronisations, and concerns the implicit interaction between the scoping convention rooted in machines and interfaces, and the scoping implications of synchronisations. This affects how we must reinterpret a single machine PO if we are to reuse it in a multi-machine context, since the events of a single machine (that might occur individually in a single machine PO) may be involved in nontrivial synchronisations.

Regarding synchronisations, our comments on their semantics in Sections 7.2 and 8 help. Suppose a synchronisation *Sync* combines events $M_1.Ev_1$ and $M_2.Ev_2$. Then: the ANY scopes of $M_1.Ev_1$ and $M_2.Ev_2$ are unified, with the replacement of I/O parameter pairs throughout by a common local parameter (e.g. replacing $a!$ and $a?$ by $a$ throughout); and the other event attributes are combined by conjunction, e.g. $grd_{Sync} \equiv grd_{M_1.Ev_1} \wedge grd_{M_2.Ev_2}$; ... ; $BApred_{Sync} \equiv BApred_{M_1.Ev_1} \wedge BApred_{M_2.Ev_2}$; and so on, yielding the corresponding attributes of the synchronisation *Sync*. (The same considerations apply irrespective of whether $M_1$ and $M_2$ are different or not.) This interpretation of the synchronisations declared within a project can now be made to correspond to the event clusters of the runtime semantics.

*We simplify things further by stipulating that pliant events are always unsynchronised.* In practice, pliant behaviour originates, almost exclusively, from some piece of identifiable physical apparatus, most naturally modelled by encapsulating it in a machine of its own, precluding the need for synchronisation. Besides, unlike mode events, pliant events do not preempt, so identifying the conditions that initiate them is relatively easy. If synchronised pliant behaviour is needed, it can be programmed simply by enabling the relevant pliant events via one or more purposely designed mode synchronisations. This simplification makes the adaptation of the single machine POs to the multi-machine/synchronisation context easier.

With this understanding of synchronisations, we now consider the interpretation of the single machine POs within the multi-machine project context in more detail. We leave till later the question of the extent to which these POs, interpreted in the way we indicate, are sufficient to ensure the correctness of multi-machine projects. We recall that an event cluster —and therefore a synchronisation, by our conflation of the two concepts here— may consist of a single event according to Section 11, and (4).

**Definition 12.3.** *Let Sync be a synchronisation in a project $\mathcal{P}$.*

*(i) A machine $M \in \mathcal{P}$ is relevant to Sync iff it contains an event of Sync.*

*(ii) An interface $I \in \mathcal{P}$ is relevant to Sync iff some machine M relevant to Sync CONNECTS to or READS I, or some machine M relevant to Sync CONNECTS to or READS an interface J which READS or REFERS to I.*

*(iii) A variable v is relevant to Sync iff it is declared in a machine or interface that is relevant to Sync.*

*(iv) An invariant Inv is relevant to Sync iff it is declared in a machine or interface that is relevant to Sync.*

Our aim is to ensure that in the multi-machine context, each single machine PO is reinterpreted in a scope wide enough to include all machines or interfaces (and in particular, the variable declarations that they contain), that may be relevant to any synchronisation that includes any event occurring in the original PO. This done, we investigate the extent to which multi-machine correctness follows.

We start by looking at single machine POs which verify some property of a single event. Reinterpreting such a PO so that it works for a (single or multi-event) synchronisation *Sync*, is straightforward.

In instantiating such a PO we use the synchronisation attributes defined above for *Sync* instead of the attributes of a single event, and the scope of the PO in terms of variables and invariants that figure in the PO becomes restricted to the variables and invariants that occur in the machines and interfaces relevant to *Sync*. This approach is sufficient for the POs: Init/FIS, Init/INV, MoEv/FIS, PliEv/FIS, MoEv/INV, PliEv/INV, MoEv/NewRV, Init/FISR.

The next easiest single machine POs to deal with are those POs that involve a single abstract event and a single concrete event that refines it (contained in a refining machine). Where there are no possible nontrivial synchronisations, the normal scoping of the PO (namely just the abstract and concrete machines containing the events, plus any relevant interfaces), is sufficient. This covers (in particular) refinement POs for pliant events: PliEv/FISR, PliEv/GRDR, PliEv/INVR, PliEv/FISRW, PliEv/INVRW.

For mode events, subject to nontrivial synchronisations, a potential complication can enter in the following manner. According to the final remarks in Section 7.2 regarding **[♦16]**–**[♦17]**, an abstract synchronisation must be refined to a corresponding concrete synchronisation. This is so that any invariants that depend on the abstract synchronicity survive at the concrete level. That however, does not prevent such a concrete synchronisation from containing additional concrete events. In particular, it introduces the possibility of concrete refining events synchronising with the events of a machine newly introduced during the refinement step (a very useful design capability). Also there may be additional synchronisations at the concrete level, that do not correspond to any abstract synchronisation.

The possibility of additional concrete events in a concrete synchronisation leads to the possibility of *additional concrete variables* (referred to in this manner below) occurring in the concrete synchronisation, which are not refinements of variables in the abstract synchronisation. This complicates the determination of appropriate scopes when abstract and concrete synchronisations both appear in a PO.

Returning to the class of POs we are interested in, namely refinement POs for mode events, we note that each of those POs *assumes* a property of the concrete event and *infers* a property of the abstract event. Reinterpreting them for the multi-machine situation, in which events become synchronisations, we see that any of the additional concrete variables that we spoke of can occur only in the assumptions, and being unconnected to any variables occurring in the PO conclusions, do no harm. Their occurrences in the assumptions are thus implicitly quantified. This covers POs: Init/INVR, MoEv/FISR, MoEv/GRDR, MoEv/INVR, MoEv/FISRW, MoEv/GRDRW, MoEv/INVRW, MoEv/NewR. Leaving aside trivial cases, the modified POs appear in (17)–(22) in Section 13 below.

The remaining single machine POs involve multiple occurrences of events in both the assumptions and conclusions. The easiest to deal with is PliEv/RelDLF, because pliant events are unsynchronised. Being unsynchronised, there is no possibility of having additional concrete variables. It is thus sufficient to reinterpret PliEv/RelDLF for multi-machine working in the context of the variables declared in the abstract and concrete machines and interfaces relevant to the abstract and concrete events (a.k.a. synchronisations) in the single machine PO. Although the manipulations are routine for this PO, to give an example of the routine case, the PO itself is quoted in (23).

The mode counterpart of the preceding is MoEv/RelDLF. As in the previous case there are multiple events in the assumptions and conclusions. However, here the situation is the converse of the one for (17)–(22), in that the concrete synchronisations are in the *conclusions* of the PO, therefore potentially mentioning additional concrete variables there. These variables must be existentially quantified. We avoid the existential quantification itself by employing a witness relation to choose the values of these additional variables appropriately, depending on the variables in the assumptions, and thus delegating the choice of values to the application design process. Besides this, each abstract synchronisation in the PO assumptions may involve a different set of additional machines, leading to a different set of concrete machines and additional concrete variables. So the argument just made must be iterated across all the synchronisations in the assumptions, and the results aggregated using one witness relation with a large number of arguments. The modified PO is (24).

Two POs remain: MoPli/WFor and PliMo/WFor. These exhibit many of the technical issues that we have met already. In MoPli/WFor there is a single mode synchronisation in the assumptions, with multiple mode synchronisations in the conclusions. The latter will, in general, introduce additional variables, not present in the assumptions (though these are not the 'additional concrete variables' spoken of above because only one level of abstraction is involved). Since we are disabling the mode synchronisations in the conclusions, their additional variables must be universally quantified (i.e. existentially quantified under the negation). The conclusions also contain pliant events, but since these are unsynchronised, the relevant variables are those of the machine(s) containing the mode synchronisation in the assumptions (together with those of any relevant interfaces). There is one further complication. If the synchronisation in the assumptions is non-trivial, it involves multiple machines, and a pliant event must be enabled in each of them (to avoid falling foul of step **[6]** in the formal semantics). Therefore, in the conclusions, the disjunction of pliant event guards (of the single machine PO) must be remodelled as a conjunction (over machines) of per-machine disjunctions of the relevant pliant events. The modified PO is (25).

In PliMo/WFor there is a single pliant event in the assumptions. The mode events of the machine containing the pliant event will, in general, participate in different synchronisations, involving different other machines and different additional variables. The conclusions of the PO contain negative occurrences (of the guards) of these synchronisations. Their purpose is to ensure that none of the synchronisations is enabled prior to preemption time. The additional variables in these occurrences must be universally quantified to achieve this. The PO also contains positive occurrences of these synchronisations. Their purpose is to ensure that at least one of them *is* enabled *at* preemption time. The additional variables in these occurrences must be existentially quantified. The modified PO is (26).

This completes the coverage of the single machine POs. We saw that the POs that introduced the greatest additional complexity were the ones that established some aspect of liveness. This is hardly surprising. It is also evident that due to the greater ambiguity concerning scopes of the various terms occurring in the POs, the multi-machine versions that we derived give the impression of offering a weaker approximation to reachability than is the case for the single machine POs.

With the above perspective established, we turn to correctness for multi-machine projects. For simplicity, we will assume without comment that all machine instantiations have been fully evaluated, so that we don't need to refer to $\mathcal{P}^I$ explicitly. This also means that all internal communications inside mode synchronisations are understood to have been remodelled as shared bound variables. This allows synchronisations to be seen as the event clusters of the runtime semantics, as noted already.[13]

**Theorem 12.4.** *Let $\mathcal{P}$ be a Hybrid Event-B project. Suppose the following conditions are satisfied.*

*(i) For every valuation $\mathcal{V}$ of all the state variables and every variable $v$, if events $M_1.Ev_1$ and $M_2.Ev_2$ both update $v$ (where machines $M_1$ and $M_2$ are distinct), then at most one of $M_1.Ev_1$ and $M_2.Ev_2$ has its WHERE guard satisfiable in $\mathcal{V}$.*

*(ii) Pliant events are unsynchronised.*

*(iii) For every pliant event PliEv, only mode variables declared in the machine that contains PliEv occur in the WHERE guard of PliEv.*

*(iv) For every pliant event PliEv, for every pliant transition determined by PliEv, if the pliant transition becomes infeasible at some $\mathbb{t}_R > \mathbb{t}_L$, then the state variable valuation at $\mathbb{t}_R$ enables a mode event in the machine that contains PliEv.*

*Suppose further that for every machine $M \in \mathcal{P}$, the single machine POs listed in Section 12.1, and reinterpreted as described earlier and in Section 13, hold. Then $\mathcal{P}$ is correct according to Definition 11.3.*

[13]In principle, event clusters could even be defined dynamically. Nothing in the runtime semantics requires a static definition.

*Proof:* Firstly, we assume as part of event feasibility, that no event or synchronisation (whether mode or pliant) has an inconsistent specification for the update of any variable. Then, it is sufficient to go through the steps of the formal semantics in Section 11 for the project $\mathcal{P}$, and to confirm that the static properties assumed are sufficient to ensure that the ABORT cases are never encountered. Once this process loops back, we have, in essence, an inductive proof of the theorem.

Regarding step **[2]**, we assume that initialisation assigns values to all variables, consistent with the invariants.

Next, the mode-to-pliant well-formedness PO MoPli/WFor applied to each machine, guarantees that for each machine, no mode event without inputs is enabled, passing step **[3]**; it also guarantees that for each machine, there is an enabled pliant event determining its subsequent behaviour, passing step **[4]**. Assumption (i) ensures that for each variable, and thus for each pliant variable, only one machine can have enabled events that govern that variable. Since only one such event can be scheduled per machine, conflicting updates to any variable in events of different machines are impossible, ensuring that step **[4.1]** is passed.

Since we are at the start of the system run, $PliEvCT = \emptyset$ in step **[5]**, and since all variables have an acceptable initial value and every pliant variable is governed by some pliant event selected in step **[4]**, step **[6]** is also passed.

Pliant event feasibility, PliEv/FIS, ensures that in step **[7]**, some nonempty interval $(t_0 \ldots t_{\mathrm{MAX}})$ can be found, leading to a choice of explicit solution for some maximal $t_{\mathrm{MAX}}$ in **[8]**. Assumption (i) ensures that in step **[8.1]** a unique $MchPliEv(pli,t)$ exists, since only one machine can update any pliant variable, and step **[8.2]** is unproblematic. The invariant preservation PO for pliant events PliEv/INV, ensures that in step **[9]**, if there is no preemption point during the pliant behaviour, then the execution continues indefinitely or becomes infeasible (giving finite termination at $t_{\mathrm{MAX}}$ in the latter case — which, although permitted by the runtime semantics, is actually prevented by assumption (iv)), so step **[9]** is passed unproblematically.

Otherwise, there is a preemption point, chosen according to the detailed criteria in **[10]**. The invariant preservation PO for pliant events PliEv/INV, ensures that step **[11]** is passed unproblematically.

Now, the next cycle of execution starts. After reindexing, **[12]**, in step **[12.1]**, assumption (i) ensures that for each variable, only one machine (for an event) or set of machines (for a synchronisation) can have enabled events that update that variable. Since only one such event or synchronisation can be scheduled per machine (or set of machines), conflicting updates to any variable in events of different machines are impossible, passing step **[12.1]**. The mode event invariant preservation PO MoEv/INV, ensures that step **[12.2]** can be passed. For step **[12.3]**, we know that the invariants hold up to, but not necessarily including $t_\eta$. Since the semantics of Hybrid Event-B is restricted to piecewise absolutely continuous behaviours in which the left and right limits exist for all times, the needed left limit exists and is finite. The closure interpretation of the invariants (if needed, see PaperI) ensures that this limit preserves the invariants. Step **[12.4]** cleans up the time interval $(t_\eta \ldots t_{\mathrm{MAX}})$ when necessary.

The proof then continues as from the third paragraph above, though it deals with a generic $t_\eta$ instead of $t_0$. The one difference between the first pliant transition and those encountered later, is that for subsequent pliant transitions, the preceding non-*INITIALISATION* mode transitions need not involve *all* of the machines $M_1 \ldots M_n$. So there are two cases: machines $M_{EN}$, that experience a mode transition, and machines $M_{CT}$, that do *not* experience a mode transition. For the latter, an ‘interrupt and resume’ behaviour should take place.

For $M_{EN}$, the argument is as above. Some mode event or synchronisation involving $M_{EN}$ was executed, and the mode-to-pliant well-formedness PO MoPli/WFor, applied to each such machine, guarantees that there is an enabled pliant event, $M_{EN}.PliEv_{NEXT}$ say, determining its subsequent behaviour.

For $M_{CT}$, we need to check the entry conditions in steps **[5]** and **[6]** hold in order to allow the ‘interrupt and resume’, and then to ensure the resumption actually takes place.

For step **[5]**, by assumption (ii), $M_{CT}$ was pursuing a pliant event of its own at $t_\eta$, say $M_{CT}.PliEv_{CT}$.

By assumption (iii), every mode variable in the WHERE guard of $PliEv_{CT}$ is declared in $M_{CT}$, beyond the reach of a mode event in any machine other than $M_{CT}$ to update. Since $M_{CT}$ did not experience a mode transition, each such variable therefore continues to have the value it had immediately preceding $t_\eta$, giving **[5]**.(4), from which we conclude that the WHERE guard of $PliEv_{CT}$ is true at $t_\eta$. Knowing that, we can choose to impose **[5]**.(3) at $t_\eta$ in $M_{CT}$ provided that $PliEv_{CT}$ is in a position to continue. To establish the latter fact, suppose that $PliEv_{CT}$, at $t_\eta$, reached left-limit values $\overrightarrow{vars(t_\eta)}$ for the variables it updates, which made $PliEv_{CT}$ infeasible beyond $t_\eta$. Then by assumption (iv), a mode event would have been enabled and would have executed at $t_\eta$ in $M_{CT}$. Since we assumed that that did not happen, the values $\overrightarrow{vars(t_\eta)}$ are in the interior of the feasible set (of variable valuations) of $PliEv_{CT}$. Furthermore, the mode event at $t_\eta$ did not update any variable updated by $PliEv_{CT}$, since to do so would have contravened assumption (i). So we deduce that $\overrightarrow{vars(t_\eta)} = vars(t_\eta)$, and since these values are in the interior of the feasible set of $PliEv_{CT}$, $PliEv_{CT}$ is able to continue.

So $M_{EN}.PliEv_{NEXT}$ and $M_{CT}.PliEv_{CT}$ are both separately enabled to continue pliant behaviour after the mode transition. If they both wished to update the same variable $v$, it would contradict assumption (i) again, from which we deduce that **[5]**.(2) holds. Since each pliant event is unsynchronised, it constitutes a pliant event cluster by itself, and thus, since all machines that did not experience the mode transition have (by induction hypothesis) a pliant event that is running, all such event clusters constitute a set of 'interrupt and resume' event clusters that is maximal, giving **[5]**.(1). For **[6]**, we observe that if any pliant variable is not explicitly mentioned in any of the pliant events that are about to execute (whether 'new' or 'resume' events), it is covered by the default COMPLY *Invariants* provision described in PaperI.

Thus we have consistent feasible behaviour specified for all the pliant variables, and the execution can resume at point **[8]**. We are done. $\square$

We extend this result to cover refinement.

**Theorem 12.5.** *Let $\mathcal{P}$ and $\mathcal{P}R$ be Hybrid Event-B projects. Suppose $\mathcal{P}R$ is a syntactic refinement of project $\mathcal{P}$ — in the sense that it is connected syntactically to $\mathcal{P}$ in the way expected for a refinement, and also obeys the restrictions described in Section 5 for refinements of projects. Suppose the conditions of Theorem 12.4 are satisfied for both projects, and additionally, the refinement POs of Section 12.1, reinterpreted as described earlier and in Section 13, hold. Then $\mathcal{P}R$ is a correct refinement of $\mathcal{P}$ according to Definition 11.5.*

*Proof:* As for single machines, the claim of the theorem states that: the abstract system trace's temporal duration is at least as long as the concrete one's; that all the invariants in both systems hold throughout the common part of the durations; and that each abstract mode event occurrence corresponds to a concrete mode event occurrence.

The proof proceeds by induction. So let *SR* be a system trace of $\mathcal{P}R$, given by a collection of time dependent valuations $y(t)$ for all the variables of all the machines over an interval $[t_0 \ldots t_{\mathrm{FIN}R})$. We show that we can simulate *SR* by a system trace *S* of $\mathcal{P}$, such that all the invariants of both projects hold during $[t_0 \ldots t_{\mathrm{FIN}R})$, and at each occurrence of a mode event in *S*, there is an occurrence of a mode event in *SR*. We reuse arguments in the proof of Theorem 12.4 whenever we can.

System trace *SR* starts with an initial state satisfying $\mathcal{P}R$'s invariants, and the initialisation refinement POs, applied per machine, ensure a corresponding $\mathcal{P}$ initial state satisfying $\mathcal{P}$'s invariants. Thereafter, pliant transitions and mode transitions alternate in *SR*.

The process starts with a pliant transition. By the pliant refinement feasibility and guard strengthening POs, PliEv/FISR and PliEv/GRDR, applied per machine, a pliant transition is enabled and feasible, and thus starts in each machine, creating a pliant transition in *S*. The starting conditions of the pliant transitions in *S* and *SR* satisfy the assumptions of the refinement invariant preservation PO PliEv/INVR, on a per machine basis. Therefore, we can deduce that as long as all the machines (concrete and abstract) are executing their pliant transitions, the invariants are maintained by $S$, provided they are maintained by $SR$. (We know the latter since $SR$ is a system trace of $\mathcal{P}R$, by Theorem 12.4.)

Suppose the next preemption time in $SR$ is $t_{next}$. For the sake of a contradiction suppose also that the abstract system trace $S$ reaches a state in which some mode event is enabled, sooner than this next $SR$ preemption time (at time $t_{soon} < t_{next}$ say). Then the mode event relative deadlock freedom PO MoEv/RelDLF would be applicable at $t_{soon}$, and thus there would be one or more mode events that were enabled at $t_{soon}$ in $SR$. These would cause a preemption in $SR$ at $t_{soon} < t_{next}$, and since there is no such preemption, the supposition that $S$ reached an abstract preempting state is incorrect. So the pliant transition continues in $S$ until $t_{next}$. (N.B. We assume that the witness relation in MoEv/RelDLF is comprehensive enough that it covers all correct possibilities for inference of concrete enabledness from abstract enabledness.)

When a pliant transition is preempted in $SR$, the argument of Theorem 12.4 shows that the preemption is correctly handled in $SR$ — one or more mode events (belonging to one or more synchronisations) are enabled, and execute, preserving the invariants. To each of the 'old' mode events among them, the mode event guard strengthening PO MoEv/GRDR is applicable. This ensures that a corresponding abstract mode event is enabled and executes via MoEv/FISR, and preserves the invariants via MoEv/INVR. Each 'new' mode event causes no change, i.e. a 'notional skip', at the abstract level, via MoEv/NewR. The argument justifying the application of these on a per-abstract/concrete-machine-pair basis depends on the independence of the retrieve relations for distinct abstract/concrete machine pairs and for distinct abstract/concrete interface pairs, guaranteed by conditions **[♦16]** and **[♦17]** of Section 5.

After the concrete mode transition, there is a concrete pliant transition in $SR$. This consists of one pliant transition per machine, since pliant events are unsynchronised. For each of them, the pliant event guard strengthening PO PliEv/GRDR ensures that a pliant event is enabled at the abstract level in each machine, which can then execute. Again, the argument can be carried through on a per-abstract/concrete-machine-pair basis because of conditions **[♦16]** and **[♦17]** in Section 5. The enabledness of the abstract pliant event for machines simulating a 'new' concrete event (i.e. executing a 'notional skip' at the abstract level), ensures that the preceding pliant transition did not become disabled precisely at the moment of preemption in such machines. The argument now continues as previously for the abstract and concrete pliant events, showing that the whole of the concrete system trace $SR$ can be simulated in the manner stated. The claims of the theorem are now easy to prove. □

In the above, for clarity, we restricted to discussing state variables, ignoring the impact of I/O parameters. In the presence of such parameters, the proofs can be extended to arrive at similar conclusions, albeit that additional assumptions are needed, concerning the availability of inputs, when they are required by system traces.

### *12.3. Type II Invariants and their Verification*

The proofs in the previous section relied on a machine's access to *both* interfaces of a tIIi in order to prove such invariants, since no distinction is made there between tIIi's and other invariants. In this section we optimise this by re-examining the cases where an event only needs to access either the local or the remote variables of a tIIi, in order to increase possibilities for separate working during verification.

Consider a tIIi, which we refer to as $(*) \equiv U(u) \Rightarrow V(v)$, where variables $u$ and $v$ belong to different interfaces. We start by considering mode events. In that context, we generically prime after-state expressions. Thus $(*')$ denotes $U(u') \Rightarrow V(v')$, and so on. We write the mode events of interest in the form $MoEvXYZ$, and their guards in the form $grd_{XYZ}$ where the meta-symbols $X,Y,Z$ belong to $\{U,V\}$. This notation means, for example, that the guard $g_{UVV}$ of the event $MoEvUVV$ mentions the variables $u,v$ (indicated by the $XY$ symbols in the subscript) and the before-after predicate $BApred_{UVV}$ of $MoEvUVV$ updates variable $v$ (indicated by the $Z$ symbol). The shorter notation $MoEvUV$ means that the guard mentions only $u$, and the update is to $v$ alone. We assume that for events *MoEvUU* and *MoEvVV*, verification would be restricted to variables $u$ and $v$ of the $(*)$ invariant respectively, requiring access only to the interface containing those variables. For events of the form *MoEvUVU* and *MoEvUVV* though, both parts of $(*)$ would participate in the verification process equally, requiring access to both the $u$ and $v$ interfaces, since both sets of variables are read. In the following, we suppress the I/O variables from the *BApred* terms for clarity.

**Theorem 12.6.** *Suppose a set of machines and interfaces satisfying **[♦1]–[♦15]** is given. Suppose that initial states satisfy the invariants, and suppose that all events preserve all type I invariants declared locally and in interfaces. Suppose also that any events that update both families of variables of type II invariants also preserve all invariants. Then the following conditions for mode events are sufficient to preserve type II invariants.*

$$MoEvUVU/MoEvVU : grd_{UVU}(u,v) \wedge \neg U(u) \wedge BApred_{UVU}(u,u') \Rightarrow \neg U(u') \tag{9}$$

$$MoEvUU : grd_{UU}(u) \wedge \neg U(u) \wedge BApred_{UU}(u,u') \Rightarrow \neg U(u') \tag{10}$$

$$MoEvUVV/MoEvUV : grd_{UVV}(u,v) \wedge V(v) \wedge BApred_{UVV}(v,v') \Rightarrow V(v') \tag{11}$$

$$MoEvVV : grd_{VV}(v) \wedge V(v) \Rightarrow BApred_{VV}(v,v') \Rightarrow V(v') \tag{12}$$

*Proof:* We focus on the inductive step of an inductive invariant preservation argument, assuming the initial states satisfy all the invariants, that we are dealing with a mode transition of the inductive argument, and that in the first part of the inductive step of the argument, all the tIi's have been taken care of. We also assume that this first part covers any events that update *both* families of variables of any tIIi (e.g. if the relevant machine connects to both interfaces). Thus the only remaining events that might violate the invariants are events that involve one or other family of variables that occur in a tIIi such as $(*)$, i.e. events of the kind mentioned. We exhibit the argument for *MoEvUVU*, treating the other cases more briefly.

Let $u, v$ be (the relevant part of) a state that is reached (by executions that are covered by the inductive-proof-so-far) and suppose $u, v$ satisfies the invariants. Suppose *MoEvUVU* is enabled and is executed. Evidently $grd_{UVU}(u,v)$ holds in the before-state. If $V(v)$ holds in the before-state then we are done, since only $u$ is updated by *MoEvUVU*, and thus $V(v') \Leftrightarrow V(v)$, whence we get $(*')$ immediately. Otherwise, $\neg V(v)$ holds, whence $grd_{UVU}(u,v) \wedge \neg V(v)$ is true, whence we get $grd_{UVU}(u,v) \wedge \neg U(u)$ by contraposition of $(*)$. Therefore $grd_{UVU}(u,v) \wedge \neg U(u) \wedge BApred_{UVU}(u,u')$ holds, the hypothesis (9). Assuming (9) therefore, we conclude $\neg U(u')$, which is enough for $(*')$.

For *MoEvVU*, the argument is the same except that $grd_{VU}$ just depends on $v$.

For the *MoEvUU* case, suppose *MoEvUU* is enabled. Then $grd_{UU}$ depends only on $u$, and based on (10), we can reuse the previous argument from the $grd_{UU} \wedge \neg U$ point.

For the *MoEvUVV* case, suppose *MoEvUVV* is enabled. If $U(u)$ does not hold then we are done, since only $v$ is updated. Otherwise, $U(u)$ holds and we conclude, in turn, $grd_{UVV} \wedge U$, $grd_{UVV} \wedge V$, and $grd_{UVV} \wedge V \wedge BApred_{UVV}$, the hypothesis of (11). Assuming (11), we conclude $V(v')$, whence $(*')$. There is the evident simplification for *MoEvUV*.

For the *MoEvVV* case, suppose *MoEvVV* is enabled. Then the above argument is again easily adapted, relying on (12). This completes all the cases, and the inductive step of the invariant preservation argument. □

Of course, examining the above proof shows that we only need the hypotheses of the theorem to hold for *reachable states*, since only those take part in any run of the system. But reachability is hard to capture in general, so the weaker formulation is much more suited to static analysis. The same remarks in fact apply to *all* argumennts based on statically constructed POs. The implications (9)-(12) thus give a range of convenient POs to be used in place of MoEv/INV when the right structural conditions obtain.

We turn to pliant events. The analogue of MoEv/INV is now PliEv/INV. The key observation is that the temporal coincidence of the before- and after- states in a mode transition is never exploited in arguments like the proof of Theorem 12.6. Therefore the same arguments can be used in the context of a pliant transition, with the previous mode before-state corresponding to the initial state of the pliant transition, and the previous mode after-state corresponding to an arbitrary non-initial state in the execution of the pliant transition. After all, the system's invariants hold in the initial state, and are required to hold for all such non-initial states to establish PliEv/INV. Thus we can conclude the following.

**Theorem 12.7.** *Suppose a set of machines and interfaces satisfying **[♦1]–[♦15]** is given. Suppose that initial states satisfy the invariants, and suppose that all events preserve all type I invariants declared locally and in interfaces. Suppose also that any events that update both families of variables of type II invariants also preserve all invariants on reachable states. Then the following conditions for pliant events are sufficient to preserve type II invariants. In (13)-(16), in keeping with the demands of PliEv/INV, t ranges over the open interval* $(\mathbb{t}_{\mathrm{L}}, \mathbb{t}_{\mathrm{R}})$*, and we have removed this quantification, and mention of I/O variables, for clarity.*

$$\begin{aligned}
PliEvUVU/PliEvVU :\ & iv_{UVU}(u(\mathbb{t}_{\mathrm{L}}),v(\mathbb{t}_{\mathrm{L}})) \wedge grd_{UVU}(u(\mathbb{t}_{\mathrm{L}}),v(\mathbb{t}_{\mathrm{L}})) \wedge \neg U(u(\mathbb{t}_{\mathrm{L}})) \wedge \\
& BDApred_{UVU}(u(\mathbb{t}_{\mathrm{L}}),u(t)) \wedge SOL_{UVU}(u(\mathbb{t}_{\mathrm{L}}),u(t)) \Rightarrow \neg U(u(t)) \qquad (13)\\
PliEvUU :\ & iv_{UU}(u(\mathbb{t}_{\mathrm{L}})) \wedge grd_{UU}(u(\mathbb{t}_{\mathrm{L}})) \wedge \neg U(u(\mathbb{t}_{\mathrm{L}})) \wedge \\
& BDApred_{UU}(u(\mathbb{t}_{\mathrm{L}}),u(t)) \wedge SOL_{UU}(u(\mathbb{t}_{\mathrm{L}}),u(t)) \Rightarrow \neg U(u(t)) \qquad (14)\\
PliEvUVV/PliEvUV :\ & iv_{UVV}(u(\mathbb{t}_{\mathrm{L}}),v(\mathbb{t}_{\mathrm{L}})) \wedge grd_{UVV}(u(\mathbb{t}_{\mathrm{L}}),v(\mathbb{t}_{\mathrm{L}})) \wedge V(v(\mathbb{t}_{\mathrm{L}})) \wedge \\
& BDApred_{UVV}(u(\mathbb{t}_{\mathrm{L}}),u(t)) \wedge SOL_{UVV}(u(\mathbb{t}_{\mathrm{L}}),u(t)) \Rightarrow V(v(t)) \qquad (15)\\
PliEvVV :\ & iv_{VV}(v(\mathbb{t}_{\mathrm{L}})) \wedge grd_{VV}(v(\mathbb{t}_{\mathrm{L}})) \wedge V(v(\mathbb{t}_{\mathrm{L}})) \wedge \\
& BDApred_{VV}(v(\mathbb{t}_{\mathrm{L}}),v(t)) \wedge SOL_{VV}(v(\mathbb{t}_{\mathrm{L}}),v(t)) \Rightarrow V(v(t)) \qquad (16)
\end{aligned}$$

## 13. Proof Obligations

In this section we present in detail the modified and additional POs discussed in the previous section for the verification of multi-machine projects. For clarity, we refer to events as *Mch.Ev*, quoting a machine identifier to enable different machines to be distinguished. Where different synchronisations need to be distinguished, we use a synchronisation identifier: *Sync.Ev*; and where we need both we can use both: *Sync.Mch.Ev*. PO (23) is an example of a routine PO, where no additional variables of any kind need to be considered, other cases of which are omitted.

We start with the refinement POs for mode events. The easiest is modified refinement feasibility MoEv/FISR, since the corresponding abstract event does not figure in the PO. In (17) we introduce notational conventions that will be maintained in the rest of this section.

$$\begin{aligned}
& \exists u;x \bullet K(u,w;x,z) \wedge grd_{SyncC.MoEvC}(w,j?,k;z) \\
& \quad \Rightarrow (\exists w',p!;z' \bullet BApred_{SyncC.MoEvC}(w,j?,k,p!,w';z,z')) \qquad (17)
\end{aligned}$$

In (17), compared with its single machine precursor, aside from the renaming of the event from *MoEvC* to *SyncC.MoEvC*, we have the additional state variables $z$ (with after-values $z'$). These belong to any enlargement of the concrete synchronisation *SyncC* that extends the set of events that *SyncC* contains beyond the set of refinements of the events in the abstract synchronisation *SyncA* that it refines according to the conditions in **[♦16]–[♦17]**. The additional state variables $z$ may come with additional parameters for the enlarged synchronisation, but we do not write these separately, in order to save clutter. The variables $z$ are separated from the earlier variables by a semicolon, to highlight their different origin.

Also, they will have counterparts $x$ in the abstract project (mentioned in the retrieve relation $K$ in (17)), unless $z$ belongs to machine(s) freshly introduced during the refinement.

The next PO is the modified guard strengthening PO, MoEv/GRDR. With the notational conventions established, this becomes:

$$\begin{aligned}&I(u) \wedge K(u,w;x,z) \wedge grd_{SyncC.MoEvC}(w,j?,k;z)\\ &\quad\Rightarrow (\exists i?,l \bullet grd_{SyncA.MoEvA}(u,i?,l))\end{aligned} \tag{18}$$

In (18), although the abstract counterpart $x$ of the concrete additional variable $z$ appears in $K$, it does not appear in data for *SyncA.MoEvA* since the abstract events in *SyncA.MoEvA* do not involve $x$.

The modification of invariant preservation MoEv/INVR now follows straightforwardly:

$$\begin{aligned}&I(u) \wedge K(u,w;x,z) \wedge grd_{SyncC.MoEvC}(w,j?,k;z) \wedge BApred_{SyncC.MoEvC}(w,j?,k,p!,w';z,z')\\ &\quad\Rightarrow (\exists i?,l,o!,u' \bullet BApred_{SyncA.MoEvA}(u,i?,l,o!,u') \wedge K(u',w';x,z'))\end{aligned} \tag{19}$$

The next three POs just reprise the preceding in the presence of a witness function and are given without further comment:

$$\begin{aligned}&I(u) \wedge K(u,w;x,z) \wedge grd_{SyncC.MoEvC}(w,j?,k;z) \wedge BApred_{SyncC.MoEvC}(w,j?,k,p!,w';z,z')\\ &\quad\Rightarrow (\exists i?,l,o!,u' \bullet W(u,i?,l,o!,u',w,j?,k,p!,w';z,z'))\end{aligned} \tag{20}$$

$$\begin{aligned}&I(u) \wedge K(u,w;x,z) \wedge grd_{SyncC.MoEvC}(w,j?,k;z) \wedge W(u,i?,l,o!,u',w,j?,k,p!,w';z,z')\\ &\quad\Rightarrow grd_{SyncA.MoEvA}(u,i?,l)\end{aligned} \tag{21}$$

$$\begin{aligned}&I(u) \wedge K(u,w;x,z) \wedge grd_{SyncC.MoEvC}(w,j?,k;z) \wedge BApred_{SyncC.MoEvC}(w,j?,k,p!,w';z,z') \wedge\\ &W(u,i?,l,o!,u',w,j?,k,p!,w';z,z')\\ &\quad\Rightarrow BApred_{SyncA.MoEvA}(u,i?,l,o!,u') \wedge K(u',w';x,z')\end{aligned} \tag{22}$$

We examine the relative deadlock freedom POs next. For pliant events, which are unsynchronised, we merely prefix each event identifier with the relevant machine name, *MchA* or *MchC*, where the latter refines the former:

$$\begin{aligned}&I(u) \wedge K(u(\mathbb{t}_{\mathrm{L}}),w(\mathbb{t}_{\mathrm{L}})) \wedge\\ &[\ (iv_{MchA.PliEvA1}(u(\mathbb{t}_{\mathrm{L}})) \wedge grd_{MchA.PliEvA1}(u(\mathbb{t}_{\mathrm{L}}))) \vee\\ &\ (iv_{MchA.PliEvA2}(u(\mathbb{t}_{\mathrm{L}})) \wedge grd_{MchA.PliEvA2}(u(\mathbb{t}_{\mathrm{L}})) \vee \ldots \vee\\ &\ (iv_{MchA.PliEvAM}(u(\mathbb{t}_{\mathrm{L}})) \wedge grd_{MchA.PliEvAM}(u(\mathbb{t}_{\mathrm{L}}))\ ]\\ &\quad\Rightarrow [\ (iv_{MchC.PliEvC1}(w(\mathbb{t}_{\mathrm{L}})) \wedge grd_{MchC.PliEvC1}(w(\mathbb{t}_{\mathrm{L}}))) \vee\\ &\qquad (iv_{MchC.PliEvC2}(w(\mathbb{t}_{\mathrm{L}})) \wedge grd_{MchC.PliEvC2}(w(\mathbb{t}_{\mathrm{L}})) \vee \ldots \vee\\ &\qquad (iv_{MchC.PliEvCN}(w(\mathbb{t}_{\mathrm{L}})) \wedge grd_{MchC.PliEvCN}(w(\mathbb{t}_{\mathrm{L}}))\ ]\end{aligned} \tag{23}$$

The more interesting case is for mode event relative deadlock freedom. As explained in Section 12.2, each abstract event might belong to a different synchronisation, leading to different additional variables at the concrete level. The witness relation takes care of connecting all the appropriate values together:

$$\begin{aligned}&I(u) \wedge K(u,w) \wedge (\exists o!,p!,u',w' \bullet W(u,i?,l,o!,u',w,j?,k,p!,w';z_1,z_2 \ldots z_M)) \wedge\\ &[\ grd_{SyncA1.MchA.MoEvA1}(u,i?,l) \vee grd_{SyncA2.MchA.MoEvA2}(u,i?,l) \vee \ldots \vee\\ &\ grd_{SyncAN.MchA.MoEvAN}(u,i?,l)\ ]\\ &\quad\Rightarrow grd_{SyncC1.MchC.MoEvC1}(w,j?,k;z_1) \vee grd_{SyncC2.MchC.MoEvC2}(w,j?,k;z_2) \vee \ldots \vee\\ &\qquad grd_{SyncCM.MchC.MoEvCM}(w,j?,k;z_M)\end{aligned} \tag{24}$$

Last, we turn to the well formedness POs. In MoPli/WFor (25), we suppose that there are $K$ machines involved in the synchronisation in the assumptions, *Sync.MoEv*. We suppose that there are $N$ machines involved in the synchronisations to be disabled at the value $u$, these being all the machines that in one of their synchronisations, include a mode event from one of the machines needed for *Sync.MoEv*. The variables $z_1 \ldots z_N$ are the variables belonging to those $N$ machines (and relevant interfaces) that are *not* involved in the synchronisation *Sync.MoEv* itself. We suppose that the $K$ machines of *Sync.MoEv* have $M_1, M_2 \ldots M_K$ pliant events respectively that need to be enabled on an at-least-one-per-machine basis.

$$
\begin{aligned}
&\exists u_0, i_0?, l_0, o_0! \bullet I(u_0) \wedge grd_{Sync.MoEv}(u_0, i_0?, l_0) \wedge BApred_{Sync.MoEv}(u_0, i_0?, l_0, o_0!, u) \wedge I(u) \\
&\quad \Rightarrow \neg [\, \exists l, z_1, z_2 \ldots z_N \bullet \\
&\qquad\qquad grd_{Sync1.MoEv1}(u, l, z_1) \vee grd_{Sync2.MoEv2}(u, l, z_2) \vee \ldots \vee grd_{SyncN.MoEvN}(u, l, z_N) \,] \wedge \\
&\qquad [\, (iv_{Mch1.PliEv1}(u) \wedge grd_{Mch1.PliEv1}(u)) \vee (iv_{Mch1.PliEv2}(u) \wedge grd_{Mch1.PliEv2}(u)) \vee \ldots \vee \\
&\qquad\; (iv_{Mch1.PliEvM_1}(u) \wedge grd_{Mch1.PliEvM_1}(u)) \,] \wedge \\
&\qquad [\, (iv_{Mch2.PliEv1}(u) \wedge grd_{Mch2.PliEv1}(u)) \vee (iv_{Mch2.PliEv2}(u) \wedge grd_{Mch2.PliEv2}(u)) \vee \ldots \vee \\
&\qquad\; (iv_{Mch2.PliEvM_2}(u) \wedge grd_{Mch2.PliEvM_2}(u)) \,] \wedge \ldots \wedge \\
&\qquad [\, (iv_{MchK.PliEv1}(u) \wedge grd_{MchK.PliEv1}(u)) \vee (iv_{MchK.PliEv2}(u) \wedge grd_{MchK.PliEv2}(u)) \vee \ldots \vee \\
&\qquad\; (iv_{MchK.PliEvM_K}(u) \wedge grd_{MchK.PliEvM_K}(u)) \,]
\end{aligned} \tag{25}
$$

In PliMo/WFor (26), since the assumptions involve a single, unsynchronised, pliant event *PliEv*, it belongs to a single machine, *Mch* say. The mode events of *Mch* may belong to a number of non-trivial synchronisations, involving a number of variables $z_1, z_2 \ldots z_N$ that are not declared in *Mch* or any interface relevant to *PliEv*. These figure in the disabling of the synchronisations prior to $\mathbb{t}_{\mathrm{R}}$ and the enabling of at least one at $\mathbb{t}_{\mathrm{R}}$.

$$
\begin{aligned}
&I(u(\mathbb{t}_{\mathrm{L}})) \wedge iv_{PliEv}(u(\mathbb{t}_{\mathrm{L}})) \wedge grd_{PliEv}(u(\mathbb{t}_{\mathrm{L}})) \wedge (\exists \mathbb{t}_{\mathrm{R}} > \mathbb{t}_{\mathrm{L}} \bullet (\forall \mathbb{t}_{\mathrm{L}} \le t < \mathbb{t}_{\mathrm{R}}, u(t), i?(t), l(t), o!(t) \bullet \\
&BDApred_{PliEv}(u(t), i?(t), l(t), o!(t), t) \wedge SOL_{PliEv}(u(t), i?(t), l(t), o!(t), t) \wedge \mathsf{MAXIMAL}(\mathbb{t}_{\mathrm{R}}) \wedge \\
&\neg [\, \exists i?, l, z_1, z_2 \ldots z_N \bullet grd_{Sync1.MoEv1}(u(t), i?, l, z_1) \vee grd_{Sync2.MoEv2}(u(t), i?, l, z_2) \vee \ldots \vee \\
&\qquad\qquad grd_{SyncN.MoEvN}(u(t), i?, l, z_N) \,])) \\
&\quad \Rightarrow \mathsf{WELLDEF}(\mathbb{t}_{\mathrm{R}}) \wedge [\, \exists i?, l, z_1, z_2 \ldots z_N \bullet \\
&\qquad\qquad grd_{Sync1.MoEv1}(\overrightarrow{u(\mathbb{t}_{\mathrm{R}})}, i?, l) \vee grd_{Sync2.MoEv2}(\overrightarrow{u(\mathbb{t}_{\mathrm{R}})}, i?, l) \vee \ldots \vee \\
&\qquad\qquad grd_{SyncN.MoEvN}(\overrightarrow{u(\mathbb{t}_{\mathrm{R}})}, i?, l) \,]
\end{aligned} \tag{26}
$$

Besides the modifications to single machine POs that we have just presented, additional conditions arise from the verification of multi-machine projects. Theorem 12.4 imposes four conditions that need to be checked in order that the soundness result follows. We consider them one by one.

Condition (i) says the following. For all variables $u$ of an interface $I$, for all values of $u$, for all event pairs from different machines that are connected to $I$ and that can update $u$, the conjunction of the WHERE guards of the pair is false. Writing the lexical assumptions to the left of a turnstile, this yields:

$$
Mch1 \neq Mch2 \wedge \mathsf{declared\_in\_interface}(u, Itf) \;\vdash\; grd_{Mch1.Ev1}(u) \wedge grd_{Mch2.Ev2}(u) \Rightarrow \mathsf{false} \tag{27}
$$

A lot can be done to soften the impact of (27). Static and *functional* assignments of variables to the machines whose events can update them are very simple to check, and can dramatically simplify the task of confirming (27).

Condition (ii) states that pliant events are unsynchronised. This imposes a simple lexical/syntactic condition to check.

Condition (iii) states that every mode variable in the WHERE guard of any pliant event *PliEv* is declared within the same machine as *PliEv*. This is another simple lexical/syntactic condition to check.

Condition (iv) states that for every pliant event *PliEv*, for every pliant transition determined by *PliEv*, if the pliant transition becomes infeasible at some $\mathbb{t}_R > \mathbb{t}_L$, then the state variable valuation at $\mathbb{t}_R$ enables a mode event in the machine that contains *PliEv*. This can be read as a variation on PliMo/WFor (26), replacing $\textsf{MAXIMAL}(\mathbb{t}_R)$ with $\textsf{INFEASIBLE}(\mathbb{t}_R)$, which not only asserts that $\mathbb{t}_R$ is maximal, as previously, but that $BDApred_{PliEv} \wedge SOL_{PliEv}$ is infeasible beyond $\mathbb{t}_R$. We do not repeat the PO in full.

And, besides all the above, we must take into account the repercussions of the optimisation of tIIi's, illustrated in Theorem 12.6 and Theorem 12.7, where an implementation chooses to use the optimisation. The impact of the tIIi structure on other POs than the ones discussed in Theorem 12.6 and Theorem 12.7 may be readily inferred from the proofs of those theorems.

## 14. Conclusions

In the preceding sections, we started by briefly reviewing the Hybrid Event-B formalism for single machines developed in detail in PaperI, [9], and used the ideas there as a springboard for the design of a Hybrid Event-B formalism for multiple cooperating machines. These typically arise through the decomposition of a more complex design, or through the instantiation/composition of existing components into a desired architecture. Our approach to that design problem was guided by a number of principles.

First and foremost was the need for important invariants to survive any structuring process. To achieve this, project partitioning is made subordinate to the structure and arrangement of the invariants, and the hypergraph project architecture pattern gives guidance on how the best partitioning may be arrived at. Acknowledging that the structure and arrangement of invariants is not guaranteed to align with architectural goals, type II invariants give additional scope for compromising between desired architecture and verified properties. Of course, it is possible that further patterns for invariants might prove necessary in some cases, but experience with the hypergraph architecture tends to suggest this might not be so likely. The provision of global invariants in the project file may prove to be all that is needed.

A second guiding consideration was the need for project structure to not introduce any semantic surprises. Thus the semantics of a multi-machine project should be easily related to the semantics of a corresponding single machine. In fact the soundness of our semantic definition for multi-machine projects was established precisely by relating their behaviours to those of associated single machines.

We gave a couple of simple case studies to illustrate the methodology. The first, concerned with power switching, featured decomposition and refinement, but restricted to mode events. The second was a decomposed presentation of the European Train Control System case study, originally done as a single machine in PaperI. We commented that larger case studies, elsewhere, had provided the fuel for the development of the the hypergraph project architecture pattern.

As noted in Section 11, the semantics of multi-machine projects was a more bureaucratic version of the single machine case, the increasing complexity coming from the way that multi-machine projects, with their variables shared via interfaces and synchronisations, could potentially interfere with one another. The main additional issue was the need for different machines to be able to undergo mode events independently, an unavoidable complication if real world behaviour is to be seriously addressed. This required some care to arrive at a definition that was compatible enough with the single machine view. This done, we were able to discuss how the single machine proof obligations, discussed in detail in PaperI, could be reinterpreted in the multi-machine context, to give a verification strategy for the multi-machine case, culminating in the simplifications for type II invariants.

All of this has yielded an expressive formal modelling and verification framework for hybrid and cyberphysical systems. Future work will concentrate on the details of the reasoning facilities needed to support this framework in an appropriate extension of the Rodin tool system [3, 21, 17].

## References

[1] J.R. Abrial, Event-B: Structure and Laws, in: Rodin Project Deliverable D7: Event-B Language. http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf.

[2] J.R. Abrial, Modeling in Event-B: System and Software Engineering, Cambridge University Press, 2010.

[3] J.R. Abrial, M. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, L. Voisin, Rodin: An Open Toolset for Modelling and Reasoning in Event-B, STTT 12 (2010) 447–466.

[4] L. de Alfaro, T.A. Henzinger, Interface Automata, in: Proc. 8th ESEC/FSE-9, ACM, 2001, pp. 109–120.

[5] R. Banach, Pliant Modalities in Hybrid Event-B, in: Liu, Woodcock, Zhu (Eds.), Proc. Jifeng He Festschrift 2013, volume 8051 of *LNCS*, Springer, 2013, pp. 37–53.

[6] R. Banach, Invariant Guided System Decomposition, in: Ait Ameur, Schewe (Eds.), Proc. ABZ-14, volume 8477 of *LNCS*, Springer, 2014, pp. 271–276.

[7] R. Banach, The Landing Gear Case Study in Hybrid Event-B, in: Ait Ameur, Schewe (Eds.), Proc. ABZ-14: Landing Gear Case Study, volume 433 of *CCIS*, Springer, 2014, pp. 126–141.

[8] R. Banach, The Landing Gear System in Multi-Machine Hybrid Event-B, Int. J. Soft. Tools for Tech. Trans. (2015). In press.

[9] R. Banach, M. Butler, S. Qin, N. Verma, H. Zhu, Core Hybrid Event-B I: Single Hybrid Event-B Machines, Sci. Comp. Prog. 105 (2015) 92–123.

[10] R. Banach, P. Van Schaik, E. Verhulst, Simulation and Formal Modelling of Yaw Control in a Drive-by-Wire Application, in: Ganzha, Maciaszek, Paprzycki (Eds.), Proc. FedCSIS IWCPS-15, pp. 731–742.

[11] F. Boniol, V. Wiels, The Landing Gear System Case Study, in: Proc. ABZ-14: Landing Gear Case Study, volume 433 of *CCIS*, Springer, 2014, pp. 1–18.

[12] M. Butler, Decomposition Strategies for Event-B, in: Leuschel, Wehrheim (Ed.), Proc. IFM-09, volume 5423, Springer, LNCS, 2009, pp. 20–38.

[13] R. Eshuis, Reconciling Statechart Semantics, Sci. Comp. Prog. 74 (2009) 65–99.

[14] S. Hallerstede, T. Hoang, Refinement by Interface Instantiation, in: Derrick, Fitzgerald, Gnesi, Khurshid, Leuschel, Reeves, Riccobene (Ed.), Proc. ABZ-12, volume 7316, Springer, LNCS, 2012, pp. 223–237.

[15] T. Hoang, J.R. Abrial, Event-B Decomposition for Parallel Programs, in: Frappier, Glässer, Khurshid, Laleau, Reeves (Ed.), Proc. ABZ-10, volume 5977, Springer, LNCS, 2010, pp. 319–333.

[16] R. Leadbetter, S. Cambanis, V. Pipiras, A Basic Course in Measure and Probability, Cambridge University Press, 2014.

[17] RODIN Tool. http://www.event-b.org/ http://www.rodintools.org/ http://sourceforge.net/projects/rodin-b-sharp/.

[18] H. Royden, P. Fitzpatrick, Real Analysis, Pearson, 2010.

[19] R. Silva, M. Butler, Supporting Reuse of Event-B Developments through Generic Instantiation, in: Breitman, Cavalcanti (Ed.), Proc. ICFEM-09, volume 5885, Springer, LNCS, 2009, pp. 466–484.

[20] R. Silva, C. Pascal, T. Hoang, M. Butler, Decomposition Tool for Event-B, Software Practice and Experience 41 (2011) 199–208.

[21] L. Voisin, J.R. Abrial, The Rodin Platform Has Turned Ten, in: Ait Ameur, Schewe (Ed.), Proc. ABZ-14, volume 8477, Springer, LNCS, 2014, pp. 1–8.

[22] W. Walter, Ordinary Differential Equations, Springer, 1998.

[23] Wikipedia, Absolute continuity.