# On the purpose of Event-B proof obligations

Stefan Hallerstede

Under consideration for publication in Formal Aspects of Computing

# On the purpose of Event-B proof obligations

Stefan Hallerstede

University of Düsseldorf, Germany

**Abstract.** Event-B is a formal modelling method which is claimed to be suitable for diverse modelling domains, such as reactive systems and sequential program development. This claim hinges on the fact that any particular model has an appropriate semantics. In Event-B this semantics is provided implicitly by proof obligations associated with a model. There is no fixed semantics though. In this article we argue that this approach is beneficial to modelling because we can use similar proof obligations across a variety of modelling domains. By way of two examples we show how similar proof obligations are linked to different semantics. A small set of proof obligations is thus suitable for a whole range of modelling problems in diverse modelling domains.

**Keywords:** Event-B, Proof Obligation, Proof, Semantics

## 1. Introduction

Event-B [Abr08] is a formal modelling method for discrete systems based on refinement [AH07, AM98, Bac89]. We believe that formal modelling should serve primarily for reasoning. We insist that reasoning is an essential part of modelling because it is the key to understanding complex models. The formal text making up a formal model should be stated in a form that facilitates reasoning. Reasoning about complex models should not happen accidentally but needs systematic support within the modelling method. This thinking lies at the heart of the Event-B method.

When we create a complex model, usually, our understanding of it is incomplete at first; and a modelling method should help to improve our understanding of the model. During initial phases in the modelling process refinement is used to manage the many details of a complex model. It does not describe a development process where we follow prescribed stages when building a model but a technique to introduce detail gradually at a rate that eases understanding. Besides, this approach yields a higher degree of automation; smaller batches of information are easier to analyse. We do not assume that we have one most abstract model, the specification, that could serve as point of reference for all further refinements. Instead, the model is completed by refinement until we are satisfied that the model captures all important properties. Eventually, we also reason about specific computational domains like sequential or concurrent programs that have well-known semantics. We expect that a formal modelling method leads safely to a correct implementation.

Covering such a range of modelling problems poses a challenge on the method. In Event-B the challenge is met by focusing solely on proof obligations that are associated with a model. The meaning of an Event-B model emerges from what is proved about the model. This gives a prominent rôle to proof obligations, the subject of this article.

*Correspondence and offprint requests to*: Stefan Hallerstede, Institut für Informatik, University of Düsseldorf, Universitätsstraße 1, 40225 Düsseldorf, Germany, e-mail: stefan.hallerstede@wanadoo.fr

### 1.1. Proof Obligations

Proof obligations are at the heart of the Event-B method. The main tool in Event-B for reasoning is formal proof. What is to be proved is stated in terms of proof obligations of a model. Event-B associates with a model proof obligations similarly to other formal methods, e.g., [Abr96, Bac89, WD96] but much attention is paid to the need to deal with frequent changes during modelling. Proof obligations serve to verify properties of a model; they serve to demonstrate that a model is sound with respect to some behavioural semantics; they serve to analyse a model; they serve to guide the user while building a model. We consider the latter point to be of major importance: when the user fails to discharge a proof obligation, usually, the corresponding proof attempt provides a hint how to improve the model. It gives the user the opportunity to gain more insight into the model and improve it gradually as the understanding increases. When creating complex models, we certainly make mistakes and we certainly have to make frequent changes to a model. This concerns the entire model across all levels of refinement, from the very abstract to the very concrete.

We assume that there is a software tool [ABHV06] that automatically generates proof obligations. Otherwise making changes to a model would be tedious and in models with thousands of proof obligations [BA05] nearly impossible. The tool takes the formal text of the model and produces proof obligations composed of (usually) only gently rewritten fragments of that text. The proof obligations of Event-B are specifically designed to permit matching them easily to the formal model [Hal05]. We illustrate this by means of a small example:

**Example.** We present an excerpt of a model of a secure building. The details of the Event-B notation are not of importance but only the way model and proof obligation match. The model has two variables $in$ and $auth$, specifying locations of persons in rooms and corresponding authorisations. The invariant of the model is:

| | |
|---|---|
| $inv1 \;:\; auth \in Person \leftrightarrow Room$ | A person is authorised to be in certain rooms |
| $inv2 \;:\; in \in Person \nrightarrow Room$ | A person can be at most in one room |
| $inv3 \;:\; in \subseteq auth$ | A person can only be in rooms where he is authorised to be |

An event $enter$ models a user entering a room:

event $enter$
  any
    $u$, $r$
  when
    $grd1 \;:\; u \notin \mathrm{dom}(in)$
    $grd2 \;:\; u \mapsto r \in auth$
  then
    $act1 \;:\; in := in \cup \{u \mapsto r\}$
  end

For the model to be consistent we prove that event $enter$ respects the invariants $inv1$, $inv2$, $inv3$. The corresponding proof obligation for $inv3$ is:

| | |
|---|---|
| $auth \in Person \leftrightarrow Room$ | invariant $inv1$ |
| $in \in Person \nrightarrow Room$ | invariant $inv2$ |
| $in \subseteq auth$ | invariant $inv3$ |
| $u \notin \mathrm{dom}(in)$ | guard $grd1$ |
| $u \mapsto r \in auth$ | guard $grd2$ |
| $\vdash\ in \cup \{u \mapsto r\} \subseteq auth$ | modified ($act1$) invariant $inv3$ |

It is very easy to relate the proof obligation to the model above. Modelling in Event-B relies entirely on the interplay between editing models and analysing their proof obligations. To the user of Event-B proof obligations appear as giving meaning to a model.

In Event-B we focus on the proof obligations and do not present a behavioural semantics at all. This approach permits us to use the same proof obligations for very different modelling domains, for instance: reactive, distributed and concurrent systems [ACM03], a probabilistic variant [HH07]; sequential programs [Abr03]; or digital circuits [Hal03]. All of this, without being constrained to a behavioural semantics tailored to a particular domain. Event-B is a calculus for modelling that is independent of the various models of computation. In the following, we use the term semantics in the sense of behavioural semantics.

### 1.2. Semantics

The core of this article deals with the problem of how an Event-B model receives a meaning. When developing a sequential program, for instance, we want to know whether the program is correct, performs a meaningful computation. We need to prove the right facts about the program; and the right facts are determined by sequential program semantics, e.g., [AO91]. But we do not want to tie Event-B to some particular semantics as explained in Section 1.1.

From the various semantics of the different modelling domains sound proof obligations have evolved for each domain. All we do is to exploit the similarities between them. To ensure that the proof obligations of Event-B can be used with common semantic models mentioned above, they have been derived from a simple relational semantic model that is not restrictive [Abr08]. When applying Event-B to a specific modelling problem we can use instead of the simple model whatever semantic model is suitable for the modelling problem at hand. This is supported without passing through some predetermined semantics —not even the simple model— of Event-B that could add some complications to the endeavour by having to relate to it.

In this article we present two examples of Event-B semantics showing the viability of this approach. For this purpose, we introduce enabledness proof obligations into the Event-B method. We go on to show how they are incorporated into relative deadlock-freeness proofs with respect to the failures model of CSP [Hoa85] and into soundness proofs of sequential program development [Abr03]. The theoretical results as such are not new. For instance, in [AM98] a temporal *leadsto*-operator and deadlock-freeness are introduced, where the *leadsto*-operator is modelled by means of a while-loop. In this article we discuss the use of the same (few) proof obligations to reason about different semantic models. We present the derivation of the proof obligations from the semantic models to demonstrate what is involved. The theory used in this article is more based on [BvW98, RE98] than on [Abr96, Abr99b]; the latter articles are geared towards sequential program development.

A complication arises (by our choice), because the first semantics uses a relational model [HJ98] and the second set transformers [BvW98, Dij76]. This complication is hidden in Event-B by means of its proof obligations: to the user of Event-B it all looks the same. Simple restrictions on proof obligations achieve soundness in either case. Because Event-B models do not have a (behavioural) semantics a priori, we are free to choose one and with it a set of appropriate proof obligations. If we were to fix some semantics for Event-B, we would have difficulties applying it to the various domains mentioned above.

### 1.3. Outline

Section 2 presents Event-B in terms of its proof obligations. In Sections 3 and 4 we relate a reactive systems semantics and a sequential program semantics to proof obligations presented in Section 2. Sections 3 and 4 are somewhat technical. We have chosen to present the material in this way to demonstrate how enabledness proof obligations arise in the two cases. As a consequence of this decision there is no space to present more examples. It is not our intention to present a complete list of semantics for Event-B. That list is open-ended. In future, new applications of Event-B may emerge that require new kinds of semantics. In the same sense, the two examples presented are not intended be understood as fully representing the corresponding domains, reactive systems modelling and sequential program modelling. The two seem reasonable based on our experience. They could be adapted to fit particular modelling needs and development processes. Whenever we want to use Event-B with some specific semantics we can prove how Event-B suits that semantics. Section 5 contains an example of a sequential program development to illustrate the use of Event-B. Practical use gets somewhat out of sight in Sections 3 and 4 although this is the primary concern of the Event-B method. In Section 6 we discuss limitations and possible problems with the approach to modelling taken by Event-B. We assume familiarity with basic set-theoretic notation. Less common concepts have been collected in Appendix A.

## 2. Event-B

We present the core of Event-B in terms of its proof obligations concerned with refinement and consistency. For the purposes of this article the proof obligations are only stated as set-theoretic expressions based on the simple relational model [Abr08]. In order to make them easier to digest we introduce some rudimentary notation of Event-B and define all employed sets and relations based on the notation.

Behavioural aspects of Event-B models are expressed by means of *machines*. A machine $M$ may contain *variables*, *invariants*, *events*, and *variants*. Variables $v$ define the state of a machine. They are constrained by invariants $I(v)$. (Variables occurring free in a formula are indicated in parentheses following the formula.) Possible state changes

are described by means of events $E_m$, for $m \in \alpha M$. (In the following sections it will prove useful to have events associated with indices drawn from finite sets $\alpha M$. We introduce them here to achieve a more coherent presentation.) Each event $E_m$ is composed of a *guard* $G_m(v)$ and an *action* $v :| S_m(v, v')$.[1] The guard of an event states the necessary condition under which the event may occur, and the action describes how the state variables evolve when the event occurs. We denote an event $E_m$ by

$$\mathsf{when}\ G_m(v)\ \mathsf{then}\ v :| S_m(v, v')\ \mathsf{end} \quad .$$

If the guard of an event $E_m$ is true, we denote it by

$$\mathsf{begin}\ v :| S_m(v, v')\ \mathsf{end} \quad .$$

A dedicated event with true as its guard and $v :| A(v')$ as its action is used for *initialisation*. (The predicate $A(v')$ does not refer to unprimed variables.)

The action $v :| S_m(v, v')$ describes the relationship between the state just before the action has occurred (represented by unprimed variable names $v$) and the state just after the action has occurred (represented by primed variable names $v'$). In practice, the action of an event is specified as a list of actions (for example, see Section 5) of the form

$$x := B(v) \tag{1}$$
$$x :\in B(v) \tag{2}$$
$$x :| Q(v, x') \quad , \tag{3}$$

where $x$ are some variables, $B(v)$ is an expressions, and $Q(v, x')$ a predicate. Form (1) assigns $x$ to a value $B(v)$, form (2) assigns $x$ to an element of a set $B(v)$, and form (3) assigns to $x$ a value satisfying a predicate. The first two are defined in terms of the third,

$$x := B(v) \quad \widehat{=} \quad x :| x' = B(v)$$
$$x :\in B(v) \quad \widehat{=} \quad x :| x' \in B(v) \quad ,$$

and for the third we define its before-after predicate to be

$$Q(v, x') \quad .$$

Variables occurring on the left-hand side of different actions of an event must be disjoint. All actions of an event $A(v)$ occur simultaneously which is expressed by conjoining their before-after predicates, yielding a predicate $X(v, x')$. Variables $y$ that do not appear on the left-hand side of an assignment of an action are not changed by the action. Formally, this is achieved by conjoining $X(v, x')$ with $y' = y$, yielding the complete action $v :| S_m(v, v')$ of the event.

We define sets and relations corresponding to all of the above:

$$\begin{aligned} \Phi &\ \widehat{=}\ \{v \mid \top\}\ ^{2} \\ i &\ \widehat{=}\ \{v \mid I(v)\} \\ g_m &\ \widehat{=}\ \{v \mid G_m(v)\} \\ s_m &\ \widehat{=}\ \{v \mapsto v' \mid S_m(v, v')\} \\ a &\ \widehat{=}\ \{v' \mid A(v')\} \quad , \end{aligned}$$

where $\Phi$ denotes the entire state space.

## 2.1. Machine Consistency

For each event $E_m$ of a machine $M$, *feasibility* must be proved:

$$i \cap g_m \quad \subseteq \quad s_m^{-1}[\Phi] \quad . \tag{4}$$

[1] In order to simplify the main part of this article, we do not present local variables of events here. For a detailed description of Event-B see [AH07].

[2] $\Phi$ is the Cartesian product of the types $\Delta_1, \Delta_2, \ldots, \Delta_\varkappa$ of the variables $v_1, v_2, \ldots, v_\varkappa$. Writing $\{v \mid \top\}$ we avoid introducing the component types $\Delta_1, \Delta_2, \ldots, \Delta_\varkappa$.

By proving feasibility, we ensure that $S_m$ provides an after state whenever $G_m$ holds. This means that the guard indeed represents the enabling condition of the event.

Invariants are supposed to hold whenever variable values change. Obviously, this does not hold a priori for any combination of events and invariants and, thus, needs to be proved. The corresponding proof obligation is called *invariant preservation*:

$$(g_m \lhd s_m)[i] \quad \subseteq \quad i \quad . \tag{5}$$

Similar proof obligations are associated with the initialisation event of a machine: feasibility of initialisation is $a \neq \varnothing$ and invariant establishment is $a \subseteq i$.

## 2.2. Machine Refinement

*Machine refinement* provides a means to introduce more details about the dynamic properties of a model [AH07]. For more on the well-known theory of refinement, we refer to the Action System formalism [Bac89] that has inspired the development of Event-B.

A machine $N$ can refine at most one other machine $M$. We call $M$ the *abstract* machine and $N$ a *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *gluing invariant* $J(v, w)$, where $v$ are the variables of the abstract machine and $w$ the variables of the concrete machine.[3]

Let $E_m$, for $m \in \alpha M$, be the abstract events; and let $F_n$, for $n \in \alpha N$, with $\alpha N$ a finite set and $\alpha M \subseteq \alpha N$, be the concrete events of the form:

$$\mathsf{when}\ H_n(w)\ \mathsf{then}\ w :| T_n(w, w')\ \mathsf{end} \quad ;$$

and let $w :| B(w')$ be the action of the initialisation.

The corresponding set-theoretic definitions are:

$$\begin{array}{rcl} \Psi & \widehat{=} & \{w \mid \top\} \\ k & \widehat{=} & \{v \mapsto w \mid I(v) \wedge J(v, w)\} \\ j & \widehat{=} & \{v \mapsto w \mid J(v, w)\} \\ h_n & \widehat{=} & \{w \mid H_n(w)\} \\ t_n & \widehat{=} & \{w \mapsto w' \mid T_n(w, w')\} \\ b & \widehat{=} & \{w' \mid B(w')\} \quad . \end{array}$$

Each event $E_m$ of the abstract machine is *refined* by a concrete event $F_m$. Somewhat simplified, we can say that $F_m$ refines $E_m$ if the guard of $F_m$ is stronger than the guard of $E_m$, and the gluing invariant $J(v, w)$ establishes a simulation of $F_m$ by $E_m$:

$$k\,;(h_m \lhd t_m) \quad \subseteq \quad (g_m \lhd s_m)\,;j \quad . \tag{6}$$

The corresponding proof obligation for the initialisation is $b \subseteq j[a]$. Using (5) we can infer from (6)

$$k\,;(h_m \lhd t_m) \quad \subseteq \quad (g_m \lhd s_m)\,;k \quad . \tag{7}$$

In the course of refinement, *new events* can be introduced into a model. New events must be proved to refine the implicit abstract event $skip$ that does nothing; that is, its guard is $\mathsf{true}$ and its action is $v :| v' = v$. In the notation used in this article new events are just those with indices drawn from the set $\alpha N \setminus \alpha M$.

### *Convergence*

Moreover, it may be proved that new events do not collectively diverge by means of a well-founded relation $r$. We refer to the corresponding proof obligation as *progress*:

$$k\,;(h_n \lhd t_n) \quad \subseteq \quad k\,;r \quad . \tag{8}$$

A common choice for $r$ is $\{w \mapsto w' \mid V(w) \geq 0 \wedge V(w') < V(w)\}$ where $V(w)$ is an integer expression, called *variant*, of $N$. We call events that satisfy (8) *convergent*.

[3] This explains why $N$ can refine at most one abstract machine. In Event-B the concrete machine $N$ contains the predicate $J(v, w)$ that links $N$ to the abstract machine $M$. In many formalisms the linking predicate is separated from the machines it links [RE98].

*Enabledness*

Using (4) we infer from (6),

$$k \rhd h_m \quad \subseteq \quad g_m \lhd k \quad , \tag{9}$$

the guard of the abstract event may be strengthened during refinement. As a consequence, it is sufficient if the guard of the concrete event is false, that is, $h_m = \varnothing$. This means we could refine any abstract event by a concrete event with false as its guard. Such an event can never occur. If we strengthen the guard less extremely, we still have a concrete event that may occur less often than its abstract counterpart. If this is not intended we need also to weaken the guard as discussed in the next paragraph.

Let $m \in \alpha M$ and $L \subseteq \alpha N$. We may prove that whenever the abstract machine may continue by means of event $E_m$ with guard $G_m$ then the concrete machine may continue by means of some $F_\ell$ for some $\ell \in L$:

$$k[g_m] \quad \subseteq \quad (\bigcup \ell \cdot \ell \in L \mid h_\ell) \quad . \tag{10}$$

By convention we assume that the guard $h_m$ of the concrete event that refines $E_m$ is contained in the union on the right hand side, that is, $m \in L$. If $L = \{m\}$, then combining (9) and (10) yields the equivalence of abstract guards to concrete guards under the (gluing) invariant:

$$g_m \lhd k \quad = \quad k \rhd h_m \quad .$$

If $L$ contains a new event, the relationship gets more complicated; enabledness and convergence interact. This becomes apparent in our presentation of sequential programs later. In our presentation of reactive systems below this is less visible due to some simplifications that we have made to keep it brief.

## 3. Reactive Systems Modelling

We base our presentation of reactive systems modelling on the semantics of the process algebra CSP [Hoa85, Ros88]. CSP was developed specifically for modelling of such systems [HJ98]. Its semantics is expressed in terms of finite and infinite traces, failures, and divergences describing the behaviour of a system. We focus on failures: failures refinement guarantees that we cannot introduce new deadlocks in a refined model. In Event-B this is achieved by enabledness (10). In this section we show how failures and enabledness are connected. The principle of this connection is not new [But96, Mor90]. For this reason, we only present the essential formal ingredients and proofs. We assume that the machines are free of divergences, proved by means of (8), and that all events are image-finite, that is, $\mathsf{finite}(s_m[g_m])$. As a consequence, the behaviour of machines can be described purely in terms of failures, the component most relevant to our analysis of enabledness proof obligations.

### 3.1. Failure Semantics

We define failures directly in the set-theoretic notation of Section 2; similarly to [Fis97]. Let $M$ be a machine with initialisation $a$ and events with guards $g_m$ and actions $s_m$.

For machine $M$ and a sequence of event indices $t$ we define the path of $t$ by

$$\begin{array}{rcl} \mathsf{path}_M(\langle\rangle) & \mathrel{\widehat{=}} & a \lhd \mathsf{id}_\Phi \\ \mathsf{path}_M(t^\frown\langle m\rangle) & \mathrel{\widehat{=}} & \mathsf{path}_M(t)\,;(g_m \lhd s_m) \quad . \end{array}$$

A path describes the state transition corresponding to the occurrence of $t$. If the path of $t$ is not empty, then $t$ belongs to the behaviour of $M$; we say such a $t$ is a trace of $M$. Failures are defined in terms of paths and of refusals introduced next. Being in a state satisfying some refusal $R$, none of the events indexed by $R$ can occur,

$$\mathsf{refusal}_M(R) \quad \mathrel{\widehat{=}} \quad (\bigcap m \cdot m \in R \mid \Phi \backslash g_m) \quad .$$

Failures are traces combined with refusals; the pair $(t \mapsto R)$ is a failure of $M$ if $t$ is a trace of $M$ and after having engaged in $t$ machine $M$ may be in a state where all events indexed by $R$ are refused,

$$(t \mapsto R) \in \mathsf{failure}_M \quad \mathrel{\widehat{=}} \quad \mathsf{path}_M(t) \rhd \mathsf{refusal}_M(R) \neq \varnothing$$

Failure semantics as defined here does not deal with fairness.

### 3.2. Failure Refinement

Let $C = \alpha N \backslash \alpha M$ be the indices of all new events, and for a trace $t$ and a set of event names $L$ let $t{\uparrow}L$ be $t$ with all event names in $L$ removed. We say machine $N$ failure-refines machine $M$,

$$(t \mapsto R \cup C) \in \mathsf{failure}_N \quad \Rightarrow \quad (t{\uparrow}C \mapsto R) \in \mathsf{failure}_M \quad ,$$

if the failures of $N$ are contained in the failures of $M$ modulo the new events $C$. Note, that this definition of failure refinement is not standard. We have combined the plain refinement notion of [Hoa85] with hiding of new events in order to shorten the presentation. The given refinement notion is still monotonic because hiding is monotonic. We do not suggest that this is the notion of failures refinement one should be using in practice but believe that it is sufficient to make our point about using Event-B for failure refinement of machines. A variant of it has been used to model introduction of local channels in stated based reactive models [But96].

Failure-refinement is proved by relating traces and failures of the two machines [But96]. Assume, by means of (7), we have

$$\mathsf{path}_N(t) \quad \subseteq \quad \mathsf{path}_M(t{\uparrow}C)\,;k \quad . \tag{11}$$

We observe

$$\begin{array}{lll}
 & (t \mapsto R \cup C) \in \mathsf{failure}_N & \{\ \text{def. of failure}\ \} \\
\equiv & \mathsf{path}_N(t) \rhd \mathsf{refusal}_N(R \cup C) \neq \varnothing & \{\ \text{by (11)}\ \} \\
\Rightarrow & \mathsf{path}_M(t{\uparrow}C)\,;k \rhd \mathsf{refusal}_N(R \cup C) \neq \varnothing & \{\ \text{set theory}\ \} \\
\Rightarrow & \mathsf{path}_M(t{\uparrow}C) \rhd k^{-1}[\mathsf{refusal}_N(R \cup C)] \neq \varnothing & \{\ \text{see (12) below}\ \} \\
\Rightarrow & \mathsf{path}_M(t{\uparrow}C) \rhd \mathsf{refusal}_M(R) \neq \varnothing & \{\ \text{def. of failure}\ \} \\
\equiv & (t{\uparrow}C \mapsto R) \in \mathsf{failure}_M &
\end{array}$$

that $N$ failure-refines $M$, provided

$$k^{-1}[\mathsf{refusal}_N(R \cup C)] \quad \subseteq \quad \mathsf{refusal}_M(R) \tag{12}$$

holds. We observe:

$$\begin{array}{lll}
 & k^{-1}[\mathsf{refusal}_N(R \cup C)] \ \subseteq\ \mathsf{refusal}_M(R) & \{\ \text{def. refusal}\ \} \\
\equiv & k^{-1}[(\bigcap n \cdot n \in (R \cup C) \mid \Psi\backslash h_n)] \ \subseteq\ (\bigcap m \cdot m \in R \mid \Phi\backslash g_m) & \{\ \text{set theory}\ \} \\
\equiv & k[(\bigcup m \cdot m \in R \mid g_m)] \ \subseteq\ (\bigcup n \cdot n \in (R \cup C) \mid h_n) & \{\ \text{set theory}\ \} \\
\equiv & (\bigcup m \cdot m \in R \mid k[g_m]) \ \subseteq\ (\bigcup n \cdot n \in (R \cup C) \mid h_n) & \{\ \text{set theory}\ \} \\
\equiv & \forall m \cdot m \in R \Rightarrow (k[g_m] \ \subseteq\ (\bigcup n \cdot n \in (R \cup C) \mid h_n)) & \{\ \text{set theory}\ \} \\
\Leftarrow & \forall m \cdot m \in R \Rightarrow (k[g_m] \ \subseteq\ (\bigcup n \cdot n \in (\{m\} \cup C) \mid h_n)) \quad . &
\end{array}$$

Refusals are downward closed: if $R$ is a refusal and $m \in R$ then $\{m\}$ is a refusal too. Hence, the strengthening $(\bigcup b \in (R \cup C) \cdot \ldots)$ to $(\bigcup b \in (C \cup \{a\}) \cdot \ldots)$ in the last step is not as severe as it may seem. The formula

$$k[g_m] \quad \subseteq \quad (\bigcup \ell \cdot \ell \in (\{m\} \cup C) \mid h_\ell)$$

in the last step of the calculation is just proof obligation (10) with $L = \{m\} \cup C$.

When we model reactive systems in Event-B, we do not need to be aware of the failures model. The proof obligations form a barrier that shields from the details and complications of the semantic model. Given the description of Event-B in the introduction it is tempting to interpret Event-B always in the way presented in this section. After all, Event-B is a descendant of Action Systems and has been conceived to model systems. However, the semantics of Event-B is not fixed. We can think about any Event-B machine in terms of any appropriate semantics. In the next section we discuss Event-B for sequential program development — with different semantics but with similar proof obligations to those of this section.

## 4. Sequential Program Modelling

Event-B has been used for sequential program development [Abr03]. We present a soundness argument resulting from the "defect" of Event-B not to provide preconditions for events: events are guarded and block execution when

the guard is false. In sequential program refinement preconditions are more common because they lead certainly to implementable programs. This does not hold for guards. If we were to interpret event guards as preconditions the problem would disappear. (In fact, this interpretation is customary in Z [Sek93, WD96].) We need an additional proof obligation to rectify this.

Given the problem described above: Why does Event-B not support preconditions and guards? By contrast, this is supported by the B Method [Abr96] but leads to more intricate (and sometimes obscure) proof obligations. In Event-B simplicity of the proof obligations is considered of major importance. It brings two strongly related benefits: proof obligations are easy to understand, and more efficient and comprehensive tool support is possible.

In this section we present how enabledness proof obligations arise when proving loop introduction correct in Event-B. We first present some set transformer theory. In the remainder of this section we prove loop introduction correct with respect to (forward) refinement of set transformers. The enabledness proof obligation will only appear at the very end of the proof.

### 4.1. Set Transformers

The notions introduced in this section are intended to capture semantical properties of sequential programs. This should not be confounded with the actual Event-B notation that uses first-order predicate logic and set theory presented in Section 2. The model of set transformers we use follows closely the type-theoretical model of [BvW98][4]. However, instead of type theory we use set theory which is easier to relate to Event-B; see also [RE98]. State spaces are Cartesian products denoted by the letters $\Phi$ and $\Psi$ as introduced in Section 2.

*Set transformers*[5] are functions from sets to sets. Let $g$ and $\varphi$ be subsets of $V$ and $s$ a relation. In this article we make use of the following set transformers[6]:

$$\begin{array}{lcll} \lfloor g \rfloor(\varphi) & \widehat{=} & g \cap \varphi & (\textit{assertion}) \\ \lceil g \rceil(\varphi) & \widehat{=} & (\Phi \setminus g) \cup \varphi & (\textit{assumption}) \\ \lceil s \rceil(\varphi) & \widehat{=} & \{v \mid s[\{v\}] \subseteq \varphi\} \quad . & (\textit{demonic update}) \end{array}$$

For set transformers $P$ we define precondition $\mathsf{pre}(P)$ and guard $\mathsf{grd}(P)$ by

$$\begin{array}{lcl} \mathsf{pre}(P) & \widehat{=} & P(\Phi) \\ \mathsf{grd}(P) & \widehat{=} & \Phi \setminus P(\varnothing) \quad . \end{array}$$

Note, that (4) implies $i \cap \mathsf{grd}(\lceil g_m \rceil ; \lceil s_m \rceil) = i \cap g_m$ and $i \cap \mathsf{pre}(\lfloor g_m \rfloor ; \lceil s_m \rceil) = i \cap g_m$.[7] The informal description of the meaning of a guard in the beginning of Section 2 leaves us a choice for its interpretation. It can be read as an assertion or an assumption. The standard reading of Event-B is as an assumption, that is, event $E_m$ corresponds to the set transformer

$$\lceil g_m \rceil ; \lceil s_m \rceil \quad . \tag{13}$$

Based on set transformers, sequential programs are usually specified in terms of *specification statements* [BvW98, Mor94], namely,

$$\lfloor g_m \rfloor ; \lceil s_m \rceil \quad , \tag{14}$$

where $\mathsf{grd}(\lfloor g_m \rfloor ; \lceil s_m \rceil) = \Phi$ would be required as a healthiness condition [Dij76]. The two simple laws

$$\lfloor g \rfloor ; \lceil g \rceil = \lfloor g \rfloor \tag{15}$$

$$\lceil g \rceil ; \lfloor g \rfloor = \lceil g \rceil \tag{16}$$

permit us to switch between the two representations (13) and (14) in suitable contexts.

---

[4] Our presentation is based on first-order set theory instead of higher-order logic. For this reason, we use *set transformers* instead of *predicate transformers*.

[5] We use the definitions of [BvW98] over that of [Abr96] because they seem to be easier to handle during proof; to avoid a notational clash we use $\lfloor \cdot \rfloor$ instead of $\{ \cdot \}$ and $\lceil \cdot \rceil$ instead of $[ \cdot ]$.

[6] *Angelic update* $\lfloor s \rfloor(\varphi) \;\widehat{=}\; \{v \mid s[\{v\}] \cap \varphi \neq \varnothing\}$ is missing from the list. We do not need it in this article.

[7] Sequential composition ";" of set transformers is defined by $(P ; Q)(\varphi) \;=\; P(Q(\varphi))$.

### 4.2. Refinement of Set Transformers

Denoting by $\sqsubseteq$ the ordering of set transformers

$$P \sqsubseteq Q \quad \widehat{=} \quad (\forall \varphi \cdot \varphi \subseteq \Phi \;\Rightarrow\; P(\varphi) \subseteq Q(\varphi)) \quad ,$$

an extensive refinement theory can be developed for set transformers [BvW98, RE98]. For a relation $k$ let $\lceil k \rceil^{\sim}$ be the left adjoint of the set transformer $\lceil k \rceil$. It has the following simple characterisation [RE98]:

$$\lceil k \rceil^{\sim}(\varphi) \quad = \quad k[\varphi] \quad . \tag{17}$$

A set transformer $P$ is said to be *forward refined* by a set transformer $Q$, denoted by $P \sqsubseteq_k Q$, if

$$\lceil k \rceil^{\sim} ; P \quad \sqsubseteq \quad Q ; \lceil k \rceil^{\sim} \quad .$$

Taking $P$ and $Q$ to be either of the form (13) or (14), forward refinement can be rephrased in relational terms [BvW98, RE98]:

$$\lceil g \rceil ; \lceil s \rceil \sqsubseteq_k \lceil h \rceil ; \lceil t \rceil \quad \Leftrightarrow \quad k ; (h \lhd t) \subseteq (g \lhd s) ; k \tag{18}$$

$$\lfloor g \rfloor ; \lceil s \rceil \sqsubseteq_k \lfloor h \rfloor ; \lceil t \rceil \quad \Leftrightarrow \quad g \lhd k \subseteq k \rhd h \;\wedge\; g \lhd (k ; t) \subseteq s ; k \tag{19}$$

At its core refinement in Event-B corresponds to forward refinement of universally conjunctive set transformers of the form (13). This is the interpretation used in Section 3. But Event-B does not have to be interpreted in this way. This is discussed in more detail in the remainder of this section:

We want to verify that introducing a loop as described in [Abr03] in Event-B is sound. Note that because of

$$g \lhd k \subseteq k \rhd h \;\wedge\; k ; (h \lhd t) \subseteq (g \lhd s) ; k \quad \Rightarrow \quad g \lhd (k ; t) \subseteq s ; k$$

it is sufficient to prove just $g \lhd k \subseteq k \rhd h$ on top of (18) so as to obtain (19). This indicates where to begin with a theory of sequential program refinement in Event-B. Matters get complicated by the presence of while loops and associated new events. We consider only this case because the case where loops are not involved is quite trivial as we have just seen.

### 4.3. Introduction of a While Loop

Using the small theory of set transformers of Sections 4.1 and 4.2 we show that the Event-B technique of introducing while loops is sound (with respect to set transformer semantics). First we give a brief account of the correspondence of Event-B model and while loop, and of the semantics of a while loop in terms of set transformers. Next we state the main property (22) to be shown to establish soundness of while-loop introduction in Event-B. The proof of this property stretches until Section 4.3.2 where the enabledness proof obligations reappear.

Let $m \in \alpha M$ and $n \in \alpha N \backslash \alpha M$. Let $E_m$ be an abstract event, and $F_m$ and $F_n$ concrete events,

$$\begin{array}{lcll} E_m & \widehat{=} & \mathsf{when}\ G_m(v)\ \mathsf{then}\ v :|\ S_m(v, v')\ \mathsf{end} & \text{(abstract event)} \\ F_m & \widehat{=} & \mathsf{when}\ H_m(w)\ \mathsf{then}\ w :|\ T_m(w, w')\ \mathsf{end} & \text{(concrete event refining } E_m\text{)} \\ F_n & \widehat{=} & \mathsf{when}\ H_n(w)\ \mathsf{then}\ w :|\ T_n(w, w')\ \mathsf{end} \quad . & \text{(new concrete event)} \end{array}$$

Our aim is to prove that the abstract event $E_m$ is refined by a loop composed of the new event $F_n$ followed by an assignment, the action of the concrete event $F_m$:

$$\begin{array}{l} \mathsf{when}\ A_m\ \mathsf{then} \\ \quad \mathsf{while}\ C_n\ \mathsf{do} \\ \quad\quad T_n \\ \quad \mathsf{end}\,; \\ \quad T_m \\ \mathsf{end} \end{array} \quad , \tag{20}$$

where $H_n = A_m \wedge C_n$ and $H_m = A_m \wedge D_m$. Let

$$\begin{array}{rcl} a_m & \widehat{=} & \{w \mid A_m(w)\} \\ c_n & \widehat{=} & \{w \mid C_n(w)\} \\ d_m & \widehat{=} & \{w \mid D_m(w)\} \quad . \end{array}$$

Thus, $h_n = a_m \cap c_n$ and $h_m = a_m \cap d_m$. We model the loop by the least fix point $(\mu X \cdot B(X))$, where the body $B(X)$ of the loop is given in terms of the new event $F_n$:

$$B(X) \quad \widehat{=} \quad (\lceil c_n \rceil ; \lceil t_n \rceil ; X) \sqcap \lceil \Psi \backslash c_n \rceil \quad .^{8}$$

The semantics of the term (20) is thus given by the set transformer:

$$\lfloor a_m \rfloor ; (\mu X \cdot B(X)) ; \lceil t_m \rceil \quad . \tag{21}$$

### *4.3.1. Soundness of While-Loop Introduction*

Our aim is to show that the refinement condition (22) follows from the proof obligations of Event-B (that we have supposedly discharged). We assume refinement, convergence, and enabledness have been proved. The proof to follow shows, in particular, where enabledness comes into play.

$$\lfloor g_m \rfloor ; \lceil s_m \rceil \quad \sqsubseteq_k \quad \lfloor a_m \rfloor ; (\mu X \cdot B(X)) ; \lceil t_m \rceil \quad , \tag{22}$$

*Proof of (22)*

We assume event $F_m$ refines event $E_m$,

$$\lceil g_m \rceil ; \lceil s_m \rceil \sqsubseteq_k \lceil h_m \rceil ; \lceil t_m \rceil \quad , \tag{23}$$

and the loop $(\mu X \cdot B(X))$ forward refines $skip$, that is,

$$\lceil \mathsf{id}_\Phi \rceil \quad \sqsubseteq_k \quad (\mu X \cdot B(X)) \quad , \tag{24}$$

Note, that the update $\lceil \mathsf{id}_\Phi \rceil$ does not diverge, hence, the refinement (24) requires the new concrete event $F_n$ to be convergent. Now,

$$\begin{array}{ll} & (22) \\ \equiv & \quad \{ \text{ (17) and def. of } \lfloor\text{-}\rfloor \text{ and } \sqsubseteq_k \} \\ & \lfloor g_m \rfloor ; \lceil s_m \rceil \quad \sqsubseteq_k \quad \lfloor k[g_m] \rfloor ; \lfloor a_m \rfloor ; (\mu X \cdot B(X)) ; \lceil t_m \rceil \\ \equiv & \quad \{ (\dagger) \} \\ & \lfloor g_m \rfloor ; \lceil s_m \rceil \quad \sqsubseteq_k \quad \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lceil t_m \rceil \\ \equiv & \quad \{ (\ddagger) \} \\ & \lfloor g_m \rfloor ; \lceil s_m \rceil \quad \sqsubseteq_k \quad \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lceil h_m \rceil ; \lceil t_m \rceil \\ \equiv & \quad \{ (15) \} \\ & \lfloor g_m \rfloor ; \lceil g_m \rceil ; \lceil s_m \rceil \quad \sqsubseteq_k \quad \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lceil h_m \rceil ; \lceil t_m \rceil \\ \Leftarrow & \quad \{ \lfloor g_m \rfloor \sqsubseteq_k \lfloor k[g_m] \rfloor \} \\ & (23) \wedge (24) \end{array}$$

The inference marked by (†) holds if the guard $G_m$ of the abstract event $E_m$ is preserved in $A_m$; we require a weaker form of enabledness (10):

$$k[g_m] \subseteq a_m \quad . \tag{25}$$

[8] The operator $\sqcap$ denotes *demonic choice* of set transformers: $(P \sqcap Q)(\varphi) = P(\varphi) \cap Q(\varphi)$.

We keep this in mind and continue the proof. Only the inference marked by (‡) is missing. We close the gap by proving the following claim

$$\lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lceil h_m \rceil \quad = \quad \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) \quad , \tag{26}$$

permitting us to eliminate the guard $h_m$ of the concrete event from the left hand side.

In order to eliminate $\lceil h_m \rceil$, propagating some information through the loop seems a good idea. Hence, we have a closer look at the set transformer $\lfloor k[g_m] \rfloor ; (\mu X \cdot B(X))$. Showing that (A) $k[g_m]$ is a loop invariant and (B) the loop establishes $\Psi \backslash c_n$ we get:

$$\lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) \quad = \quad \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lfloor (\Psi \backslash c_n) \cap k[g_m] \rfloor \quad . \tag{27}$$

*Proof of (27)*

Assuming the new event is convergent —as we do by (24)— we can exchange the least against the greatest fix point [BvW98, HJ98]:

$$\lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) \quad = \quad \lfloor k[g_m] \rfloor ; (\nu X \cdot B(X))$$

So we can carry out fix point calculations using the greatest fix point.

(A) We show that the image of the abstract guard under the simulation $k[g_m]$ is a loop invariant:

$$k[g_m] \subseteq (\nu X \cdot B(X))(k[g_m])$$

We know that the image of abstract guard $g_m$ is an invariant of the concrete action $s_n$ because the concrete event refines $skip$:

$$k[g_m] \quad \subseteq \quad \lceil t_n \rceil (k[g_m]) \quad . \tag{28}$$

We state without proof (compare [BvW98, Lemma 21.9], for instance):

$$(\nu X \cdot B(X))(\varphi) \quad = \quad (\nu x \cdot (c_n \cap \lceil t_n \rceil (x)) \cup ((\Psi \backslash c_n) \cap \phi)) \quad . \tag{29}$$

We prove that $k[g_m]$ is an invariant of the loop $(\nu X \cdot B(X))$. We calculate:

$$\begin{array}{lll} & (\nu X \cdot B(X))(k[g_m]) & \{ \text{ (29) } \} \\ = & (\nu x \cdot (c_n \cap \lceil t_n \rceil (x)) \cup ((\Psi \backslash c_n) \cap k[g_m])) & \{ \text{ see def. of } b(x) \text{ below } \} \\ = & (\nu x \cdot b(x)) & \{ \text{ see below } \} \\ \supseteq & k[g_m] \quad . & \end{array}$$

We define $b(x)$ by $b(x) \;\widehat{=}\; (c_n \cap \lceil t_n \rceil (x)) \cup ((\Psi \backslash c_n) \cap k[g_m])$ and prove the remaining claim

$$k[g_m] \subseteq (\nu x \cdot b(x)) \quad ;$$

we insert $k[g_m]$ into $b(x)$:

$$\begin{array}{lll} & b(k[g_m]) & \{ \text{ def. of } b(x) \text{ } \} \\ = & (c_n \cap \lceil t_n \rceil (k[g_m])) \cup ((\Psi \backslash c_n) \cap k[g_m]) & \{ \text{ (28) } \} \\ \supseteq & (c_n \cap k[g_m]) \cup ((\Psi \backslash c_n) \cap k[g_m]) & \{ \text{ set theory } \} \\ = & k[g_m] \quad . & \end{array}$$

Using the fix point property (e.g. [BvW98]),

$$\phi \subseteq b(\phi) \quad \Rightarrow \quad \phi \subseteq (\nu x \cdot b(x)) \quad ,$$

we conclude $k[g_m] \subseteq (\nu x \cdot b(x))$ as desired.

(B) We show that the loop establishes $\Psi \backslash c_n$

$$(\nu X \cdot B(X))((\Psi \backslash c_n) \cap \phi) = (\nu X \cdot B(X))(\phi)$$

In other words, $(\nu X \cdot B(X))$ establishes the negated guard of the concrete event; see [BvW98]:

$$\begin{array}{lll} & (\nu X \cdot B(X))((\Psi\backslash c_n) \cap \phi) & \{\ (29)\ \} \\ = & (\nu x \cdot (c_n \cap \lceil t_n \rceil(x)) \ \cup \ ((\Psi\backslash c_n) \cap (\Psi\backslash c_n) \cap \phi)) & \{\ \text{set theory}\ \} \\ = & (\nu x \cdot (c_n \cap \lceil t_n \rceil(x)) \ \cup \ ((\Psi\backslash c_n) \cap \phi)) & \{\ (29)\ \} \\ = & (\nu X \cdot B(X))(\phi) \quad . \end{array}$$

### *4.3.2. Use of Enabledness*

Finally, enabledness enters the picture. We can discharge (26), using (27) and enabledness.

*Proof of (26)*

We prove,

$$\begin{array}{lll} & \lfloor k[g_m] \rfloor \,;(\mu X \cdot B(X)) \,; \lceil h_m \rceil & \{\ (27)\ \} \\ = & \lfloor k[g_m] \rfloor \,;(\mu X \cdot B(X)) \,; \lfloor (\Psi\backslash h_n) \cap k[g_m] \rfloor \,; \lceil h_m \rceil & \{\ (*)\ \} \\ = & \lfloor k[g_m] \rfloor \,;(\mu X \cdot B(X)) \,; \lfloor (\Psi\backslash h_n) \cap k[g_m] \rfloor & \{\ (27)\ \} \\ = & \lfloor k[g_m] \rfloor \,;(\mu X \cdot B(X)) \quad , \end{array}$$

To fill in the gap $(*)$ we have to show

$$\lfloor (\Psi\backslash c_n) \cap k[g_m] \rfloor \,; \lceil h_m \rceil \quad = \quad \lfloor (\Psi\backslash c_n) \cap k[g_m] \rfloor$$

We already require $k[g_m] \subseteq a_m$, see (25). Hence, because $h_m = a_m \cap d_m$, it suffices to show

$$\lfloor (\Psi\backslash c_n) \cap k[g_m] \rfloor \,; \lceil d_m \rceil \quad = \quad \lfloor (\Psi\backslash c_n) \cap k[g_m] \rfloor \quad ,$$

which holds if $k[g_m] \subseteq d_m \cup c_n$. Finally,

$$\begin{array}{lllll} & k[g_m] & \subseteq & a_m \ \wedge \ k[g_m] \subseteq d_m \cup c_n & \{\ \text{set theory}\ \} \\ \equiv & k[g_m] & \subseteq & (a_m \cap d_m) \cup (a_m \cap c_n) & \{\ h_m = a_m \cap d_m \text{ and } h_n = a_m \cap c_n\ \} \\ \equiv & k[g_m] & \subseteq & h_m \cup h_n & \end{array}$$

which corresponds to the enabledness proof obligation (10) with $L = \{m, n\}$. Using this proof obligation, we have proved something about preconditions. If we were committed to the failures semantics of Event-B, we would have had difficulties seeing this. Intuitively, deadlock-freeness appears quite distant from preconditions. The enabledness proof obligations permits us to weaken preconditions as usual in sequential program refinement [Mor94]; we have $k[g_m] \subseteq h_m \cup h_n$ but only $k^{-1}[h_m] \subseteq g_m$.

Preservation of enabledness properties is achieved by simple rules governing their refinement [AM98]; the guard of each abstract event must imply the guard of the concrete event or the guard of some new event. This is just what we have shown to be necessary in this section. Loop introduction is proved by refinement. We can pull the assumption $\lceil a_m \rceil$ into the loop $B(X)$ to obtain $(\lceil a_m \rceil \,; \lceil c_n \rceil \,; \lceil t_n \rceil \,; X) \sqcap \lceil \Psi\backslash c_n \rceil$, or

$$(\lceil c_n \rceil \,; \lceil h_n \rceil \,; \lceil t_n \rceil \,; X) \sqcap \lceil \Psi\backslash c_n \rceil \quad .$$

Thus, we can refine the concrete event $F_n$ further, maybe introducing more loops. The structure of the loop guards involved is suggested by the way we have decomposed the guards of the concrete events $F_m$ and $F_n$ that are split into the precondition $A_m$ of the loop, the loop guard $C_n$, and the termination condition $D_m$ of the loop. A systematic way of deriving them is presented in [Abr03]. In Section 5 we give a brief example of a sequential program development using the same proof obligations that would apply to a reactive system development. We only need to respect the syntactic constraint concerning the guards of $F_m$ and $F_n$.

When developing sequential programs in Event-B we do not need to apply the possibly complex underlying theory directly but only know about the proof obligations of the kind given in the introduction. We do not need to be aware of the theory while modelling a program. We do not need to be aware of the theory while modelling other kinds of

system either but simply rely on the proof obligations presented to us. A large amount of those proof obligations is shared among the the different kinds of system. This makes it easy for the same person to create models in the different domains without having to learn a new approach each time.

## 5. Example

We illustrate the sequential program development in Event-B by formally developing a program that computes the factorial function. It is sufficient to get an impression of the Event-B method and it is a simple program containing a nested loop. It may be the first program for which termination has been proved by means of a lexicographical variant [MJ84, Tur49]. The purpose of this example is not to demonstrate the capacity of the Event-B method but just to fill the discussion of the method in other sections of this article with some life.

Program $fp$ below computes the factorial function for a natural number $n$ in variable $f$. As in [Tur49] the implementation does not use multiplication:

```
program fp
  v, r, s, u := 1, 0, 0, 1 ;
  while r < n do
    while s < r do
      u, s := u + v, s + 1
    end ;
    v, r, s := u, r + 1, 0
  end ;
  f := v
```

Our aim is to build a model of program $fp$. We begin by postulating the factorial function

$$axm1 : fac \in \mathbb{N} \rightarrow \mathbb{N}$$
$$axm2 : fac(0) = 1$$
$$axm3 : \forall m \cdot m > 0 \Rightarrow fac(m) = fac(m-1) * m$$

and the parameter $n$

$$axm4 : n \in \mathbb{N} \quad .$$

In the program development that follows we will only present proof obligations with most hypotheses that are not relevant removed. We have used the Rodin tool [ABHV06] to create the model and try to convey this by presenting the model as if written on a sheet with mixed elements of model and proof. We have chosen this format to suggest how models are created in Event-B in incremental steps by reasoning about different fragments of the model. We have straightened the development to achieve a more concise presentation, removing all the trial and error of the original development. Usually, the model is not written down in one step and analysed in the next step, but created piecemeal accompanied gradually by its analysis. The reasoning itself follows directly the shapes of the formulas appearing in the model [vG90]. This influences, intentionally, the form of a model and keeps the modelling effort at bay.

### 5.1. Specification of the Program

The initial model of the program only has one variable $f$

$$inv1 : f \in \mathbb{N} \quad ,$$

and uses function $fac$ postulated above to get the factorial of $n$

```
event factorial
  begin
    act1i : f := fac(n)
  end   .
```

Variable $f$ is initialised to some natural number

initialisation
  begin
    $act1 \;:\; f :\in \mathbb{N}$
  end .

We have to prove that event factorial preserves invariant $inv1$,

$$\begin{array}{ll} fac \in \mathbb{N} \to \mathbb{N} & \text{axiom } axm1 \\ n \in \mathbb{N} & \text{axiom } axm4 \\ f \in \mathbb{N} & \text{invariant } inv1 \\ \vdash\ fac(n) \in \mathbb{N} & \text{modified } (act1i) \text{ invariant } inv1 \end{array}$$

that the initialisation establishes the invariant, $f' \in \mathbb{N} \vdash f' \in \mathbb{N}$, and that the action of the initialisation is feasible, that is $\mathbb{N} \neq \varnothing$. The initial factorial program modelled is

program $fpi$
  $f := fac(n)$ .

## 5.2. Outer Loop

We introduce the outer loop of the factorial program calculating the product of the first $n$ natural numbers greater than $0$. We introduce a new variable $r$ iterating over the first $n$ natural numbers and a new variable $v$ storing the factorial of $r$,

$inv2 \;:\; r \in 0 \,..\, n$
$inv3 \;:\; v = fac(r)$ .

The proof obligation for the initialisation to establish $inv3$ imposes a constraint on the possible choices for $v$ and $r$. This constraint easily satisfied setting $v$ to $1$ and $r$ to $0$, see $axm2$. (This choice also satisfies invariant $inv2$.)

initialisation
  begin
    $act1 \;:\; f :\in \mathbb{N}$
    $act2 \;:\; v := 1$
    $act3 \;:\; r := 0$
  end .

The factorial function is refined by an event that records the final state of the computation when $r$ equals $n$:

event $factorial$
  when
    $grd1 \;:\; r = n$
  then
    $act1r \;:\; f := v$
  end .

We have to show that this refines the initial factorial event, that the abstract event $factorial$ can simulate the refined one:

$$\begin{array}{ll} v = fac(r) & \text{invariant } inv3 \\ r = n & \text{guard } grd1 \\ \vdash\ v = fac(n) & \text{modified } (act1r) \text{ action } act1i \end{array}$$ .

The loop body is modelled by the new event $mult$. Its shape is suggested by the shape of axiom $axm3$:

event $mult$
  when
    $grd1 \;:\; r < n$
  then
    $act1 \;:\; v := v * (r+1)$
    $act2 \;:\; r := r+1$
  end .

Preservation of invariant $inv2$ is easy to show; the proof obligation for the preservation of invariant $inv3$ is shown below:

| | | |
|---|---|---|
| | $\forall m \cdot m > 0 \Rightarrow fac(m) = fac(m-1) * m$ | axiom $axm3$ |
| | $r \in 0 \,..\, n$ | invariant $inv2$ |
| | $v = fac(r)$ | invariant $inv3$ |
| | $r < n$ | guard $grd1$ |
| $\vdash$ | $v * (r+1) = fac(r+1)$ | modified ($act1$, $act2$) invariant $inv3$ . |

The proof is easy, too,

| | | |
|---|---|---|
| | $fac(r+1)$ | { $axm3$ with $m := r+1$, and $inv2$ } |
| $=$ | $fac(r) * (r+1)$ | { $inv3$ } |
| $=$ | $v * (r+1)$ . | |

Inspection of event $mult$ suggests $n - r$ as variant. Guard $grd1$ implies that the variant is at least zero, $n - r \geq 0$, and event $mult$ decreases the variant, $n - (r+1) < n - r$. The enabledness proof obligation is

| | | |
|---|---|---|
| | $r \in 0 \,..\, n$ | invariant $inv2$ |
| $\vdash$ | $(r = n) \vee (r < n)$ | disjunction of the guards of $factorial$ and $mult$ . |

For now, we have arrived at the following program $fpr$

program $fp$
  $v, r := 1, 0$ ;
  while $r < n$ do
    $v, r := v * (r+1), r+1$
  end ;
  $f := v$ .

Next, we refine the body of the while loop $v, r := v * (r+1), r+1$ into another while loop.

## 5.3. Inner Loop

We implement the multiplication $v * (r+1)$ by adding $r+1$ times $v$. The intermediate result of $s+1$ additions is kept in a new variable $u$

$inv4 \;:\; u = (s+1) * v$ .

Invariant $inv4$ is satisfied setting $s$ to 0 and $u$ to 1,

initialisation
  begin
    $act1 \;:\; f :\in \mathbb{N}$
    $act2 \;:\; v := 1$
    $act3 \;:\; r := 0$
    $act4 \;:\; s := 0$
    $act5 \;:\; u := 1$
  end .

The body of the inner loop is modelled by event $add$,

event $add$
  when
    $grd1 : r < n$
    $grd2 : s < r$
  then
    $act1 : u := u + v$
    $act2 : s := s + 1$
  end .

The first guard $grd1$ is suggested by our development method as described in the beginning of Section 4.3. Using basic arithmetic we can show that event $add$ preserves invariant $inv4$,

| | | |
|---|---|---|
| | $u = (s+1) * v$ | invariant $inv3$ |
| | $r < n$ | guard $grd1$ |
| | $s < r$ | guard $grd2$ |
| $\vdash$ | $u + v = ((s+1)+1) * v$ | modified ($act1$, $act2$) invariant $inv4$ . |

Similar to the preceding refinement step, we choose $r - s$ as variant, which is at least zero by $grd2$, and is decreased by event $add$. Event $mult$ should only occur once $v$ has been added $r + 1$ times. To preserve invariant $inv4$, we have to reset $s$ to 0, refining $mult$ to

event $mult$
  when
    $grd1 : r < n$
    $grd2 : s = r$
  then
    $act1 : v := v * (r + 1)$
    $act2 : r := r + 1$
    $act3 : s := 0$
  end .

It remains to show enabledness of $mult$:

| | | |
|---|---|---|
| | $r < n$ | guards of abstract event $mult$ |
| $\vdash$ | $(r < n \wedge s = r) \vee (r < n \wedge s < r)$ | disjunction of the guards of concrete $mult$ and $add$ . |

Our invariant is to weak to prove this. The additional invariant

$inv5 : s \in 0 \mathinner{..} r$

solves the problem. This finishes our model of program $fp$ computing the factorial of natural number $n$. Using the method described in [Abr03] and in Section 4.3 we can present the model in a more familiar form of a sequential program.

In the development presented in this example we have focused our attention on proof obligations. This is typical of Event-B developments. We have left "what the program does" to our intuitive understanding of it. All effort was spent on reasoning about the model composed of invariants and events, proving properties about it, and doing some arithmetical calculations related to the factorial function. The formal reasoning had been the same had we designed a reactive system instead of a sequential program.

## 6. Liberties and Limitations

In standard situations a system model is based on a specific, usually, well-known semantics. When using Event-B, we only consider the kind of model created to be interesting, within the scope of this article, a sequential program or a reactive system. However, in these situations we do not worry too much by which means soundness was proved with respect to the proof obligations. We simply rely on the proof obligations as they are generated by some tool [ABHV06]. In standard situations we ought to be able to focus on modelling, and writing a model should become mere routine. The tool could statically determine whether some semantics is satisfied based on the discharged proof

obligations. Alternatively, syntax and proof obligations could be constrained to suit some semantics. However, it is also possible that a model does not fit one of those situations. For instance, in the model described in [Abr99a] some events that are newly introduced must be convergent and some need not be. In that case one has to be aware of the semantics of the model justifying the presence of proof obligations and the absence of proof obligations. This is the price of the liberty one can have when modelling in Event-B. We can create models unconstrained by some semantics. This may be particularly useful for experimentation. But we have to be careful about what a model means and justify why we consider a particular model reasonable. For such models the semantics can be considered to be part of the properties of the system modelled — it is no longer given a priori as is the case in standard situations.

## 7. Conclusion

Event-B addresses various modelling domains among which are the reactive systems and sequential programs presented in this article. Event-B has a notation based on first-order predicate logic and set theory. Event-B has a set of proof obligations that are associated with models.

What Event-B lacks is a behavioural semantics. And that is so intentionally. In fact, it would be difficult to support all those modelling domains using one semantics that would suit all. What we have seen, by way of two examples, is that the proof obligations of Event-B can be used in a way to fit with some intended semantics, be it relational or predicate transformer-based, be it for reactive systems or sequential programs. In some sense, in Event-B semantics is replaced by proof obligations. Possible semantics are characterised but not fixed.

The major advantage of this approach is that proof obligations can be used across the different domains. From our experience we know that they have a lot in common and it seems a good idea to exploit this. For the different domains, though, proof obligations can be proved sound with respect to appropriate semantics. Thus we would still like a model that is supposed to represent a sequential program, say, to have proof obligations that are sound with respect to a semantics for sequential programs. And this can be achieved in Event-B by linking the proof obligations to an appropriate semantic theory.

**Acknowledgment.** I want to thank Michael Butler and the anonymous reviewers for helpful remarks and suggestions that improved clarity of the presentation, and Jean-Raymond Abrial for the productive discussions on this subject. This research was carried out as part of the EU research project DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) http://www.deploy-project.eu/.

## References

[ABHV06] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM 2006*, volume 4260, pages 588–605. Springer, 2006.

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abr99a] Jean-Raymond Abrial. Event driven system construction, 1999.

[Abr99b] Jean-Raymond Abrial. Models of computations, 1999.

[Abr03] Jean-Raymond Abrial. Event based sequential program development: Application to constructing a pointer program. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 51–74. Springer, 2003.

[Abr08] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2008. To appear.

[ACM03] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of IEEE 1394 tree identify protocol. *Formal Aspects of Computing*, 14(3):215–227, 2003.

[AH07] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. *Fundamentae Informatica*, 77(1-2), 2007.

[AM98] Jean-Raymond Abrial and Louis Mussat. Introducing dynamic constraints in B. In Didier Bert, editor, *B'98 : The 2nd International B Conference*, volume 1393 of *LNCS*, pages 83–128. Springer, 1998.

[AO91] K. R. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs*. Springer-Verlag, New York, Berlin, 1991.

[BA05] Frédéric Badeau and Arnaud Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB 2005*, volume 3455 of *LNCS*, pages 334–354, 2005.

[Bac89] Ralph-Johan Back. Refinement Calculus II: Parallel and Reactive Programs. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer, May 1989.

[But96] Michael J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, 1996.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Fis97] Clemens Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *FMOODS '97*, volume 2, pages 423–438. Chapman & Hall, 1997.
[Hal03] Stefan Hallerstede. Parallel hardware design in B. In D. Bert, J. P. Bowen, S. King, and M. A. Waldén, editors, *ZB*, volume 2651 of *LNCS*, pages 101–102. Springer, 2003.
[Hal05] Stefan Hallerstede. The Event-B Proof Obligation Generator. Technical report, ETH Zürich, 2005.
[HH07] Stefan Hallerstede and Thai Son Hoang. Qualitative probabilistic modelling in event-B. In J. Davies and J. Gibbons, editors, *IFM 2007*, volume 4591 of *LNCS*, pages 293–312. Springer, 2007.
[HJ98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.
[Hoa85] C. A. R. Hoare. *Communcating Sequential Processes*. Prentice Hall, 1985.
[MJ84] F. Lockwood Morris and Cliff B. Jones. An Early Program Proof by Alan Turing. *Annals of the History of Computing*, 6(2):193–143, 1984.
[Mor90] Carroll C. Morgan. Of wp and CSP. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*, pages 319–326. Springer, 1990.
[Mor94] Carroll C. Morgan. *Programming from Specifications: Second Edition*. Prentice Hall, 1994.
[RE98] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science 47. CUP, 1998.
[Ros88] A. W. Roscoe. Unbounded nondeterminism in CSP. Technical Monograph PRG-67, Programming Research Group, Oxford University, 1988.
[Sek93] Emil Sekerinski. A calculus for predicative programming. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *MPC*, LNCS. Springer, 1993.
[Tur49] Alan M. Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines, EDSAC Inaugural Conference*, pages 67–69, Cambridge, UK, 1949. University Mathematical Laboratory.
[vG90] A. J. M. van Gasteren. *On the Shape of Mathematical Arguments*, volume 445 of *LNCS*. Springer, 1990.
[WD96] Jim Woodcock and Jim Davies. *Using Z. Specification, Refinement, and Proof*. Prentice-Hall, 1996.

## A. Set-theoretical Notation

Some less common set-theoretical notation:

*Converse "$r^{-1}$"*

$$x \mapsto y \in r^{-1} \quad \widehat{=} \quad y \mapsto x \in r$$

*Image "$r[s]$"*

$$y \in r[s] \quad \widehat{=} \quad \exists x \cdot x \in s \land x \mapsto y \in r$$

*Domain restriction "$s \lhd r$"*

$$x \mapsto y \in (s \lhd r) \quad \widehat{=} \quad x \in s \ \land \ x \mapsto y \in r$$

*Range restriction "$s \rhd r$"*

$$x \mapsto y \in (r \rhd s) \quad \widehat{=} \quad x \mapsto y \in r \ \land \ y \in s$$

*Composition "$r_1 ; r_2$"*

$$x \mapsto y \in (r_1 ; r_2) \quad \widehat{=} \quad \exists z \cdot x \mapsto z \in r_1 \ \land \ z \mapsto y \in r_2$$

*Interval "$a \mathinner{..} b$"*

$$x \in a \mathinner{..} b \quad \widehat{=} \quad a \leq x \ \land \ x \leq b$$