On the Purpose of Event-B Proof Obligations[*]

Stefan Hallerstede

University of Southampton
United Kingdom
sth@ecs.soton.ac.uk

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.

1 Introduction

Event-B [5] is a formal modelling method for discrete systems based on refinement [8,9,10]. The main purpose of creating models in Event-B is to reason about them and understand them. Reasoning about complex models should not happen accidentally but needs systematic support within the modelling method. We insist that reasoning is an essential part of modelling because it is the key to understanding complex models. When we create a complex model, usually, our understanding of it is incomplete at first; and the first duty of a modelling method is to help improve our understanding of the model.

To reason about a model we consider its proof obligations. Proof obligations have a two-fold purpose. On the one hand, they show that a model is sound with respect to some behavioural semantics. On the other hand, they serve to verify properties of the model. This goes so far that we only 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 [7], a probabilistic variant [16]; sequential programs [4]; or electronic circuits [15]. All of this, without being constrained to semantics tailored to a particular domain. Event-B is a calculus for modelling that is independent of the various models of computation.

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

[*] 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/.

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 [17] and into soundness proofs of sequential program development [4]. The theoretical results as such are not new. In [9] 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 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 [11,21] than on [1,3]; the latter are geared towards sequential program development.

A complication arises (by our choice), because the first semantics uses a relational model [18] and the second set transformers [11,13]. 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 chose 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 in the introduction.

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.

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. 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.) 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 αM. 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* $v \;:|\; S_m(v, v')$. We denote an event E_m by

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

A dedicated event that has $\texttt{true}$ 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').

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.

In order to simplify the main part of this article, we do not present local variables of events here. For the same reason we state actions as single nondeterministic assignments $v :| S_m(v, v')$. For a detailed description of events, actions and assignments see [8].

We assume familiarity with basic set-theoretic notation defining sets and relations corresponding to all of the above:

$$\begin{array}{rcl} \varPhi & \widehat{=} & \{v \mid \top\}^{1} \\ i & \widehat{=} & \{v \mid I(v)\} \\ g_m & \widehat{=} & \{v \mid G_m(v)\} \\ s_m & \widehat{=} & \{v \mapsto v' \mid S_m(v, v')\} \\ a & \widehat{=} & \{v' \mid A(v')\} \quad , \end{array}$$

where $\varPhi$ denotes the entire state space.

2.1 Machine Consistency

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

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

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*:

$$(g_m \lhd s_m)[i] \quad \subseteq \quad i \quad .^{2} \tag{2}$$

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

[1] $\varPhi$ is the Cartesian product of the types $\varDelta_1, \varDelta_2, \ldots, \varDelta_\varkappa$ of the variables $v_1, v_2, \ldots, v_\varkappa$. Writing $\{v \mid \top\}$ we avoid introducing the component types $\varDelta_1, \varDelta_2, \ldots, \varDelta_\varkappa$.

[2] $\lhd$ denotes *domain restriction*: $x \mapsto y \in (g \lhd s) \;\equiv\; x \in g \land x \mapsto y \in S$.

2.2 Machine Refinement

Machine refinement provides a means to introduce more details about the dynamic properties of a model [8]. For more on the well-known theory of refinement, we refer to the Action System formalism [10] 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.

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

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

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

The corresponding set-theoretic definitions are:

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

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:

$$k\,;(h_m \lhd t_m) \quad \subseteq \quad (g_m \lhd s_m)\,;j \quad .[3] \tag{3}$$

Using (2) we can infer from (3)

$$k\,;(h_m \lhd t_m) \quad \subseteq \quad (g_m \lhd s_m)\,;k \quad . \tag{4}$$

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 true and its action is $v :\mid v' = v$. 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*:

$$k\,;(h_n \lhd t_n) \quad \subseteq \quad k\,;r \quad . \tag{5}$$

A common choice for r is $\eta^{-1}\,;\{x \mapsto y \mid x < y\}\,;\eta$ where $\eta = (\lambda\, w \cdot w \in \mathbb{N} \mid V(w))$ and $V(w)$ an integer expression, called *variant*, of N. We call events that satisfy (5) *convergent*.

[3] The corresponding proof obligation for the initialisation is: $b \subseteq j[a]$.

Enabledness. Using (1) we infer from (3),

$$k \rhd h_m \quad \subseteq \quad g_m \lhd k \quad , \tag{6}$$

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_ℓ for some $\ell \in L$:

$$k[g_m] \quad \subseteq \quad (\bigcup \ell \cdot \ell \in L \mid h_\ell) \quad . \tag{7}$$

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 (6) and (7) 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 is 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 [17,22]. CSP was developed specifically for modelling of such systems [18]. 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 (7). In this section we show how failures and enabledness are connected. The principle of this connection is not new [12,19]. 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 (5), 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 [14]. 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 the 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,

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

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,

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

Failure semantics 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 [17] 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 [12].

Failure-refinement is proved by relating traces and failures of the two machines [12]. Assume, by means of (4), we have

$$\mathsf{path}_N(t) \quad \subseteq \quad \mathsf{path}_M(t{\uparrow}C)\,; k \quad . \tag{8}$$

We observe

$$\begin{array}{lll} & (t \mapsto R \cup C) \in \mathsf{failure}_N & \{ \text{ def. of failure } \} \\ \equiv & \mathsf{path}_N(t) \rhd \mathsf{refusal}_N(R \cup C) \neq \varnothing & \{ \text{ by } (8) \text{ } \} \\ \Rightarrow & \mathsf{path}_M(t{\uparrow}C)\,; k \rhd \mathsf{refusal}_N(R \cup C) \neq \varnothing & \{ \text{ set theory } \} \\ \Rightarrow & \mathsf{path}_M(t{\uparrow}C) \rhd k^{-1}[\mathsf{refusal}_N(R \cup C)] \neq \varnothing & \{ \text{ see } (9) \text{ below } \} \\ \Rightarrow & \mathsf{path}_M(t{\uparrow}C) \rhd \mathsf{refusal}_M(R) \neq \varnothing & \{ \text{ def. of failure } \} \\ \equiv & (t{\uparrow}C \mapsto R) \in \mathsf{failure}_M & \end{array}$$

that N failure-refines M, provided

$$k^{-1}[\mathsf{refusal}_N(R \cup C)] \quad \subseteq \quad \mathsf{refusal}_M(R) \tag{9}$$

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 (7) 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 [4]. 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 [23,24].) 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 [1] 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 [11][4]. However, instead of type theory we use set theory which is easier to relate to Event-B; see also [21]. State spaces are Cartesian products denoted by the letters Φ and Ψ as introduced in Section 2.

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

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

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

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

Note, that (1) implies $i \cap \mathsf{grd}(\lceil g_m \rceil \,;\, \lceil s_m \rceil) = i \cap g_m$ and $i \cap \mathsf{pre}(\lfloor g_m \rfloor \,;\, \lceil s_m \rceil) = i \cap g_m$. 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 assertion, that is, event E_m corresponds to the set transformer

$$\lceil g_m \rceil \,;\, \lceil s_m \rceil \quad . \tag{10}$$

Based on set transformers, sequential programs are usually specified in terms of *specification statements*[11,20], namely,

$$\lfloor g_m \rfloor \,;\, \lceil s_m \rceil \quad , \tag{11}$$

[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 [11] over that of [1] 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* $\lfloor s \rfloor(\varphi) \mathrel{\widehat{=}} \{v \mid s[\{v\}] \cap \varphi \neq \varnothing\}$ is missing from the list. We do not need it in this article.

where $\mathsf{grd}(\lfloor g_m \rfloor \,;\, \lceil s_m \rceil) = \Phi$ would be required as a healthiness condition [13]. The two simple laws

$$\lfloor g \rfloor \,;\, \lceil g \rceil \quad = \quad \lfloor g \rfloor \tag{12}$$
$$\lceil g \rceil \,;\, \lfloor g \rfloor \quad = \quad \lceil g \rceil \tag{13}$$

permit us to switch between the two representations (10) and (11) in suitable contexts.

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 [11,21]. 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 [21]:

$$\lceil k \rceil^{\sim}(\varphi) \quad = \quad k[\varphi] \quad . \tag{14}$$

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 (10) or (11), forward refinement can be rephrased in relational terms [11,21]:

$$\lceil g \rceil \,;\, \lceil s \rceil \sqsubseteq_k \lceil h \rceil \,;\, \lceil t \rceil \quad \Leftrightarrow \quad k \,;\, (h \lhd t) \subseteq (g \lhd s) \,;\, k \tag{15}$$
$$\lfloor g \rfloor \,;\, \lceil s \rceil \sqsubseteq_k \lfloor h \rfloor \,;\, \lceil t \rceil \quad \Leftrightarrow \quad g \lhd k \subseteq k \rhd h \;\wedge\; g \lhd (k \,;\, t) \subseteq s \,;\, k \tag{16}$$

At its core refinement in Event-B corresponds to forward refinement of universally conjunctive set transformers of the form (10). 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 [4] 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 (15) so as to obtain (16). 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

Let $m \in \alpha M$ and $n \in \alpha N \backslash \alpha M$. 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:

```
while $H_n$ do
   $T_n$
end ;
$T_m$          .
```

We model the loop by the least fix point $(\mu X \cdot B(X))$:

$$\lfloor g_m \rfloor ; \lceil s_m \rceil \quad \sqsubseteq_k \quad (\mu X \cdot B(X)) ; \lceil t_m \rceil \quad , \tag{17}$$

where the body of the loop is given in terms of the new event F_n [7]

$$B(X) \quad \widehat{=} \quad (\lceil h_n \rceil ; (\lfloor h_n \rfloor ; \lceil t_n \rceil) ; X) \sqcap \lceil \Psi \backslash h_n \rceil \quad .$$

(Because of law (13), we can simplify $B(X)$ to $(\lceil h_n \rceil ; \lceil t_n \rceil ; X) \sqcap \lceil \Psi \backslash h_n \rceil$. We may not want to carry out this simplification, though, if F_n is refined further. In that case we would want to replace $\lfloor h_n \rfloor ; \lceil t_n \rceil$ in the longer formula by whatever refines it. In that case we would like a more concise loop guard than just the guard $\lceil h_n \rceil$ of the new event. We are not concerned about the exact form of the loop guards here, however. A systematic way of deriving them is presented in [4].)

Proof of (17). We assume event F_m refines event E_m,

$$\lceil g_m \rceil ; \lceil s_m \rceil \sqsubseteq_k \lceil h_m \rceil ; \lceil t_m \rceil \quad , \tag{18}$$

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

$$\lceil \mathsf{id}_\Phi \rceil \quad \sqsubseteq_k \quad (\mu X \cdot B(X)) \quad , \tag{19}$$

Note, that the update $\lceil \mathsf{id}_\Phi \rceil$ does not diverge, hence, the refinement (19) requires the new concrete event F_n to be convergent. Now,

$$\begin{array}{ll}
 & (17) \\
\equiv & \quad \{ \, (14) \text{ and def. of } \lfloor \text{-} \rfloor \text{ and } \sqsubseteq_k \, \} \\
 & \lfloor g_m \rfloor ; \lceil s_m \rceil \quad \sqsubseteq_k \quad \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lceil t_m \rceil \\
\equiv & \quad \{ \, (*) \, \} \\
 & \lfloor g_m \rfloor ; \lceil s_m \rceil \quad \sqsubseteq_k \quad \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lceil h_m \rceil ; \lceil t_m \rceil \\
\equiv & \quad \{ \, (12) \, \} \\
 & \lfloor g_m \rfloor ; \lceil g_m \rceil ; \lceil s_m \rceil \quad \sqsubseteq_k \quad \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lceil h_m \rceil ; \lceil t_m \rceil \\
\Leftarrow & \quad \{ \, \lfloor g_m \rfloor \; \sqsubseteq_k \; \lfloor k[g_m] \rfloor \, \} \\
 & (18) \wedge (19)
\end{array}$$

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

Only the inference marked by $(*)$ is missing. We close the gap by proving the following claim

$$\lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) ; \lceil h_m \rceil \quad = \quad \lfloor k[g_m] \rfloor ; (\mu X \cdot B(X)) \quad , \tag{20}$$

permitting to eliminate the guard h_m of the concrete event from the left hand side. This is done in the next two sections: in Section 4.4 we prove two claims facilitating the conclusion of the proof of (20) in the ensuing Section 4.5.

4.4 Analysing the While Loop

Our aim is to establish (20). 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 $\lfloor k[g_m] \rfloor ; (\mu X \cdot B(X))$. Assuming the new event is convergent —as we do: (19)— we can exchange the least against the greatest fix point [11,18]:

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

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

First, we show $k[g_m] \subseteq (\nu X \cdot B(X))(k[g_m])$. We know that the abstract guard g_m is an invariant of the concrete action s_n because the concrete event refines $skip$:

$$k[g_m] \quad \subseteq \quad \lceil t_n \rceil (k[g_m]) \quad . \tag{21}$$

We state without proof (compare [11, Lemma 21.9], for instance):

$$(\nu X \cdot B(X))(\varphi) \quad = \quad (\nu x \cdot (h_n \cap \lceil t_n \rceil (x)) \cup ((\Psi \backslash h_n) \cap \phi)) \quad . \tag{22}$$

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

$$\begin{array}{lll}
 & (\nu X \cdot B(X))(k[g_m]) & \{\ (22)\ \} \\
= & (\nu x \cdot (h_n \cap \lceil t_n \rceil (x)) \cup ((\Psi \backslash h_n) \cap k[g_m])) & \{\ \text{see def. of } b(x) \text{ below}\ \} \\
= & (\nu x \cdot b(x)) & \{\ \text{see below}\ \} \\
\supseteq & k[g_m] \quad . &
\end{array}$$

We define $b(x)$ by $b(x) \ \widehat{=}\ (h_n \cap \lceil t_n \rceil (x)) \cup ((\Psi \backslash h_n) \cap k[g_m])$ and prove the remaining claim $k[g_m] \subseteq (\nu x \cdot b(x))$; we insert $k[g_m]$ into $b(x)$:

$$\begin{array}{lll}
 & b(k[g_m]) & \{\ \text{def. of } b(x)\ \} \\
= & (h_n \cap \lceil t_n \rceil (k[g_m])) \cup ((\Psi \backslash h_n) \cap k[g_m]) & \{\ (21)\ \} \\
\supseteq & (h_n \cap k[g_m]) \cup ((\Psi \backslash h_n) \cap k[g_m]) & \{\ \text{set theory}\ \} \\
= & k[g_m] \quad . &
\end{array}$$

Using the fix point property (e.g. [11]),

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

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

***Second,* $(\nu X \cdot B(X))((\Psi \backslash h_n) \cap \phi) = (\nu X \cdot B(X))(\phi)$.** In other words, we can also show that $(\nu X \cdot B(X))$ establishes the negated guard of the concrete event; see [11]:

$$\begin{array}{lll}
 & (\nu X \cdot B(X))((\Psi \backslash h_n) \cap \phi) & \{ (22) \} \\
= & (\nu x \cdot (h_n \cap \lceil t_n \rceil (x)) \cup ((\Psi \backslash h_n) \cap (\Psi \backslash h_n) \cap \phi)) & \{ \text{set theory} \} \\
= & (\nu x \cdot (h_n \cap \lceil t_n \rceil (x)) \cup ((\Psi \backslash h_n) \cap \phi)) & \{ (22) \} \\
= & (\nu X \cdot B(X))(\phi) \quad . &
\end{array}$$

4.5 Use of the Enabledness Proof Obligation

Combining the first and second claim of the preceding section, we have proved:

$$\lfloor k[g_m] \rfloor \,;\, (\mu X \cdot B(X)) \quad = \quad \lfloor k[g_m] \rfloor \,;\, (\mu X \cdot B(X)) \,;\, \lfloor (\Psi \backslash h_n) \cap k[g_m] \rfloor \; . \tag{23}$$

Finally, we can discharge (20) by means of (7):

$$\begin{array}{lll}
 & \lfloor k[g_m] \rfloor \,;\, (\mu X \cdot B(X)) \,;\, \lceil h_m \rceil & \{ (23) \} \\
= & \lfloor k[g_m] \rfloor \,;\, (\mu X \cdot B(X)) \,;\, \lfloor (\Psi \backslash h_n) \cap k[g_m] \rfloor \,;\, \lceil h_m \rceil & \{ (*) \} \\
= & \lfloor k[g_m] \rfloor \,;\, (\mu X \cdot B(X)) \,;\, \lfloor (\Psi \backslash h_n) \cap k[g_m] \rfloor & \{ (23) \} \\
= & \lfloor k[g_m] \rfloor \,;\, (\mu X \cdot B(X)) \quad , &
\end{array}$$

where the step marked by $(*)$ depends on

$$k[g_m] \quad \subseteq \quad h_m \cup h_n \tag{24}$$

which corresponds to the enabledness proof obligation (7) 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 permit us to weaken preconditions as usual in sequential program refinement [20]; 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 [9]; 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. If we continue in this way correctness is preserved.

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.

4.6 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 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 [6]. In standard situations we ought to be able to focus on modelling, and writing a model become mere routine. However, it is also possible that a model does not fit one of those situations. For instance, in the model described in [2] 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.

5 Conclusion

Event-B addresses various modelling domains among which 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 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 seems a good idea to exploit this. For the different domains, though, proof obligations can be proved sound with respect to appropriate semantics. 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 second anonymous reviewer for helpful remarks and suggestions that improved clarity of the presentation, and Jean-Raymond Abrial for the productive discussions on this subject.

References

1. Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Jean-Raymond Abrial. Event driven system construction, 1999.
3. Jean-Raymond Abrial. Models of computations, 1999.
4. 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.
5. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2008. To appear.
6. 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.
7. 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.
8. Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. *Fundamentae Informatica*, 77(1-2), 2007.
9. 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.
10. 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.
11. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.
12. Michael J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, 1996.
13. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
14. 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.
15. 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.
16. 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.
17. C. A. R. Hoare. *Communcating Sequential Processes*. Prentice Hall, 1985.
18. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.
19. 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.
20. Carroll C. Morgan. *Programming from Specifications: Second Edition*. Prentice Hall, 1994.
21. 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.
22. A. W. Roscoe. Unbounded nondeterminism in CSP. Technical Monograph PRG-67, Programming Research Group, Oxford University, 1988.
23. Emil Sekerinski. A calculus for predicative programming. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *MPC*, LNCS. Springer, 1993.
24. Jim Woodcock and Jim Davies. *Using Z. Specification, Refinement, and Proof*. Prentice-Hall, 1996.