Justifications for the Event-B Modelling Notation*

Stefan Hallerstede

ETH Zurich
Switzerland
halstefa@inf.ethz.ch

Abstract. Event-B is a notation and method for discrete systems modelling by refinement. The notation has been carefully designed to be simple and easily teachable. The simplicity of the notation takes also into account the support by a modelling tool. This is important because Event-B is intended to be used to create complex models. Without appropriate tool support this would not be possible. This article presents justifications and explanations for the choices that have been made when designing the Event-B notation.

1 Introduction

In this article we present an overview of the Event-B notation and provide justifications for the choices made when developing the notation. The Event-B notation is targeted at an incremental modelling style where models are found by trial and error. As such it is best explained by referring to concrete modelling problems resulting from this approach. For this reason we present the justifications as a list of problem statements. The guiding principles when designing the notation were its intended simplicity and the aim to make learning it easy. Usually notations get more complicated and less consistent as they evolve. As Event-B has evolved from classical B [1] and Action Systems [10] we were aware of this danger and took a considerable amount of time to discuss the notation.

Event-B [6] is a modelling notation and method for formal development of discrete systems based on refinement; see e.g. [1,10]. An Event-B model is associated with proof obligations that permit us to reason about it. This is essential for a modelling method: we must be able to reason about models written in it. In order to explain specific traits of modelling, we compare requirements for a programming notation to those for a modelling notation. There are some similarities between programming and modelling, and between the proof obligations to establish properties of programs and models. However, there are differences that have an impact on the notations used. The most notable difference is that for modelling we use refinement, introducing more detail in a step-wise fashion;

* This research was carried out as part of the EU research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems) http://rodin.cs.ncl.ac.uk.

whereas for programming we use verification [11] where all detail is introduced in a single step, i.e. when writing the program.

Event-B has been designed with tool support in mind and we have drawn on our experience with the tool Click'n'Prove [4] during the discussions. The user of an Event-B tool should be presented with proof obligations that (1) are not trivial and (2) can be easily related to the model.

The first point should allow the user to focus on the interesting part of a problem. Usually, the proportion of more challenging proof obligations makes only a small percentage of all proof obligations. We are aware that automated theorem provers can discharge most of the trivial proof obligations that appear when modelling systems. However, even as theorem provers improve further and get more powerful, modelling will remain difficult. The reason for this is that modelling is an exploratory activity that requires ingenuity in order to arrive at a meaningful model.

The second point is important because we consider proving properties about a model one of the major facilities to gain understanding of the model. When a proof obligation cannot be proved, it should be almost obvious what needs to be changed in the model. When modelling, we usually do not simply represent some system in a formal notation; but we learn about the system and eliminate misunderstandings, inconsistencies, and specification gaps. In particular, in order to eliminate misunderstandings, we first must develop an understanding of the system.

This article is organised as follows. Section 2 gives a brief overview of the Event-B notation. In Section 3 we discuss important points about the notation by stating a list of problems and the solutions we have chosen.

2 The Event-B Modelling Notation

Event-B [6], unlike classical B [1], does not have a fixed syntax. Instead it is a collection of modelling elements that are stored in a repository. This decision has been taken so that Event-B can be more easily extended with new constructs, say, to incorporate probability [17] or CSP [13,19]. Still, we present the basic notation for Event-B using some syntax. We proceed like this to improve legibility and help the reader remembering the different constructs of Event-B. The syntax should be understood as a convention for presenting Event-B models in textual form rather than defining a language. More reasons for this approach are discussed in Section 3.

Event-B models are described in terms of the two basic constructs *contexts* and *machines*. Contexts contain the static part of a model whereas machines contain the dynamic part. This is presented in Section 2.1. Contexts can be *extended* by other contexts and *referenced* by machines. Machines can be *refined* by machines. This is presented in Section 2.2.

The semantics of an Event-B model is characterised by *proof obligations*. In fact, 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, e.g., reactive, distributed and concurrent systems [5], sequential programs [2], electronic circuits [15], or mixed designs [8], not 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. We believe that this uniformity is a key to teaching the various aspects of systems modelling.

As a prerequisite to Section 3 which provides the justifications, Sections 2.1 and 2.2 give a brief overview of Event-B.

2.1 Contexts and Machines

Contexts provide axiomatic properties of Event-B models. They play also an important rôle in model parameterisation (see Section 3.10) and model instantiation [6] which is not discussed in detail in this article. Contexts may contain *carrier sets*, *constants*, *axioms*, and *theorems*. Carrier sets are similar to types but both, carrier sets and constants, can be instantiated as is customary in algebraic specification, e.g., [9]. Axioms describe properties of carrier sets and constants. Theorems are derived properties that can be proved from the axioms. Proof obligations associated with contexts are straightforward: the stated theorems must be proved. In this article we focus on (the more interesting) proof obligations associated with machines.

Machines provide behavioural properties of Event-B models. Machines M may contain *variables*, *invariants*, *theorems*, *events*, and *variants*. Variables v define the state of a machine. They are constrained by invariants $I(v)$. Possible state changes are described by means of events. Each event is composed of a *guard* $G(t, v)$ and an *action* $S(t, v)$, where t are *local variables* the event may contain. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. An event can be represented by the term

$$\mathsf{any}\ t\ \mathsf{where}\ G(t,v)\ \mathsf{then}\ S(t,v)\ \mathsf{end} \quad . \tag{1}$$

The short form

$$\mathsf{when}\ G(v)\ \mathsf{then}\ S(v)\ \mathsf{end} \tag{2}$$

is used if event e does not have local variables, and the form

$$\mathsf{begin}\ S(v)\ \mathsf{end} \tag{3}$$

if in addition the guard equals true. A dedicated event of the form (3) is used for *initialisation*. The action of an event is composed of several *assignments* of the form

$$x\ :=\ E(t,v) \tag{4}$$

$$x\ :\in\ E(t,v) \tag{5}$$

$$x\ :|\ Q(t,v,x') \quad , \tag{6}$$

where x are some variables, $E(t,v)$ expressions, and $Q(t,v,x')$ a predicate. Assignment form (4) is *deterministic*, the other two forms are *nondeterministic*. Form (5) assigns x to an element of a set, and form (6) assigns to x a value satisfying a predicate. The effect of each assignments can also be described by a before-after predicate:

$$BA\big(x \;:=\; E(t,v)\big) \quad \widehat{=} \quad x' \;=\; E(t,v) \tag{7}$$

$$BA\big(x \;:\in\; E(t,v)\big) \quad \widehat{=} \quad x' \;\in\; E(t,v) \tag{8}$$

$$BA\big(x \;:|\; Q(t,v,x')\big) \quad \widehat{=} \quad Q(t,v,x') \quad . \tag{9}$$

A before-after predicate describes the state just before an assignment has occurred (represented by unprimed variable names x) and the state just after the assignment has occurred (represented by primed variable names x'). All assignments of an action $S(t,v)$ occur simultaneously which is expressed by conjoining their before-after predicates, yielding a predicate $A(t,v,x')$. 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 $A(t,v,x')$ with $y' = y$, yielding the before-after predicate of the action:

$$BA\big(S(t,v)\big) \quad \widehat{=} \quad A(t,v,x') \;\wedge\; y' = y \quad . \tag{10}$$

In proof obligations we represent the before-after predicate $BA\big(S(t,v)\big)$ of an action $S(t,v)$ by directly by the predicate

$$\boldsymbol{S}(t,v,v') \quad .$$

Proof obligations serve to verify certain properties of a machine. All proof obligations in this article are presented in the form of sequents: "antecedent" $\vdash$ "succedent".

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

$$\vdash \begin{array}{l} I(v) \;\wedge\; G(t,v) \\ (\exists v' \cdot \boldsymbol{S}(t,v,v')) \quad . \end{array} \tag{11}$$

By proving feasibility we achieve that $\boldsymbol{S}(t,v,v')$ provides an after state whenever $G(t,v)$ 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*:

$$\vdash \begin{array}{l} I(v) \;\wedge\; G(t,v) \;\wedge\; \boldsymbol{S}(t,v,v') \\ I(v') \quad . \end{array} \tag{12}$$

Similar proof obligations are associated with the initialisation event of a machine. The only difference is that the invariant does not appear in the antecedent of the proof obligations (11) and (12). For brevity we do not treat initialisation differently from ordinary events of a machine. The required modifications of the concerned proof obligations are obvious. We also do not discuss deadlock freeness, see [6].

2.2 Context and Machine Relationships

Context extensions is a mechanism to introduce more detail into the axiomatic properties of a model by adding carrier sets, constants, and axioms, or simply to add derived properties in the form of theorems. There are no specific proof obligations dealing with context extension. In order to structure axiomatic properties, contexts may extend several other contexts, see [6].

Context references provide the means to access axiomatic properties from machines. A machine may reference several contexts. In that case we say the machine *sees* these contexts. Seeing more than one context is particularly useful in conjunction with decomposition [6].

Machine refinement provides a means to introduce more detail about the dynamic properties of a model [6]. For more on the well-known theory of refinement we refer to the Action System formalism that has inspired the development of Event-B [10]. We present some important proof obligations for machine refinement. As mentioned before, the user of Event-B is not presented with a behavioural model but only with proof obligations. The proof obligations describe the semantics of Event-B models.

A machine CM can refine at most one other machine AM. We call AM the *abstract* machine and CM a *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *glueing invariant* $J(v, w)$, where v are the variables of the abstract machine and w the variables of the concrete machine.

Each event ea of the abstract machine is *refined* by one or more concrete events ec. Let abstract event ea and concrete event ec be:

$$ea \quad \widehat{=} \quad \mathsf{any}\ t\ \mathsf{where}\ G(t,v)\ \mathsf{then}\ S(t,v)\ \mathsf{end} \tag{13}$$

$$ec \quad \widehat{=} \quad \mathsf{any}\ u\ \mathsf{where}\ H(u,w)\ \mathsf{then}\ T(u,w)\ \mathsf{end} \quad . \tag{14}$$

Somewhat simplified, we can say that ec refines ea if the guard of ec is stronger than the guard of ea, and the glueing invariant $J(v, w)$ establishes a simulation[1] of ec by ea:

$$\vdash \begin{array}{l} I(v) \ \wedge\ J(v,w) \ \wedge\ H(u,w) \ \wedge\ \boldsymbol{T}(u,w,w') \\ (\exists t \cdot G(t,v) \ \wedge\ (\exists v' \cdot \boldsymbol{S}(t,v,v') \ \wedge\ J(v',w'))) \quad . \end{array} \tag{15}$$

In the course of refinement usually *new events ec* are introduced into a model. New events must be proved to refine the implicit abstract event *skip* that does nothing. Moreover, it may be proved that new events do not collectively diverge by proving that a *variant* $V(w)$ is decreased by each new event:

$$\vdash \begin{array}{l} I(v) \ \wedge\ J(v,w)) \ \wedge\ H(u,w)) \ \wedge\ \boldsymbol{T}(u,w,w') \\ V(w) \in \mathbb{N} \ \wedge\ V(w') < V(w) \quad , \end{array} \tag{16}$$

where we assume that the variant expression is a natural number. It can be more elaborate [6] but this is not relevant here.

[1] More specifically, it establishes a *forward* simulation [22].

3 Modelling Problems and Solutions

In this section we present justifications for the Event-B notation, most in the form of pairs of problem statement and our solution. It resumes many discussions about the notation and method. Two major objectives during these discussions were to make the notation as simple as possible and to make learning it easy by avoiding exceptions and inconsistencies. This section is the heart of the article. The problems are not sorted according to importance. They are all important in the sense that they contribute to the overall simplicity of the Event-B notation and method.

3.1 Terminology

When writing about Event-B we found that by careless choice of terminology certain concepts are difficult to convey. For instance, instead of the word "machine", we could use "model" or "system". As a consequence, in an introductory text about B we would have phrases like: "*A model consists of models and contexts.*", or "*A system is a model of a system.*". Such phrases are a hurdle that is difficult to overcome by beginners learning Event-B. We analysed texts on Event-B, and have chosen a terminology that, we believe, is neutral with respect to modelling domains, and does not conflict with habitual modelling terminology. This is the reason why, in the end, we have chosen the word "machine" over its alternatives.

3.2 Labels

Problem Usually the invariant of a machine $I(v)$ is a conjunction of predicates

$$I_0(v) \ \wedge \ I_1(v) \ \wedge \ \ldots \ \wedge \ I_k(n) \ \wedge \ \ldots \ \wedge \ I_n(v) \quad . \tag{17}$$

When treating proof obligation (12) that serves to verify invariant preservation,

$$\vdash \begin{array}{l} I(v) \ \wedge \ G(t,v) \ \wedge \ \boldsymbol{S}(t,v,v') \\ I(v') \quad , \end{array} \tag{12}$$

we use basic sequent calculus to split the conjunction in the succedent. The aim of this is to achieve more manageable proof obligations. Instead of (12) we generate n proof obligations

$$\vdash \begin{array}{l} I(v) \ \wedge \ G(t,v) \ \wedge \ \boldsymbol{S}(t,v,v') \\ I_k(v') \quad . \end{array} \tag{18}$$

The advantage of this is that proof obligations are much smaller. The problem introduced by this technique is the following. When a model is changed it can be costly and sometimes even not possible to relate proofs belonging to previously generated proof obligations. In our concrete case of (18), if we modify the model, e.g., by inserting a predicate $I_j(v)$ into the list (17) and changing $S(t,v)$, then all

indices after the insertion point change and, at the same time, many of the predicates $I_k(v')$ of (17) change. This makes it very difficult to relate existing proofs to their associated proof obligation, stopping us from reusing them efficiently. This problem exists generally if a model contains many theorems, invariants, or events.

Solution Individual axioms, theorems, invariants, events, guards, and actions are labelled. For instance, the invariant of a model is a list of labelled predicates:

$$\begin{array}{ll} \textbf{inv\_0}: & I_0(v) \\ \textbf{inv\_1}: & I_1(v) \\ \vdots & \\ \textbf{inv\_k}: & I_k(v) \\ \vdots & \\ \textbf{inv\_n}: & I_n(v) \quad . \end{array}$$

Let e be an event with label **evt**, then instead of proof obligation (12) we generate several proof obligations (18) with names

$$\text{“}\textbf{evt}/\textbf{inv\_k}/inv\text{”} \quad ,$$

where the last segment of the name “$/inv$” depends on the proof obligation. It gives an indication about what is being proved. Now it is very easy to locate old proofs for this proof obligation by name, independently of the complexity of changes made to a model. The same approach is followed for all proof obligations associated with Event-B models. The predicates $I_k(v)$ are treated like atomic predicates during proof obligation generation so that there is an immediate correspondence between models and their proof obligations.

An additional benefit of the labels is that they can be used in the documentation of a model. They can also be useful to make the informal requirements better traceable into the model, because all Event-B modelling elements can be easily referenced by their label.

3.3 Feasibility

Problem The intention of specifying a guard of an event is that the event may always occur when the guard is true. There is, however, some interaction between guards and nondeterministic assignments (5) and (6), namely $x :\in E(t, v)$ and $x :| Q(t, v, x')$.

We say an assignment is *feasible* if there is an after-state satisfying the corresponding before-after predicate. The first form (5) is not feasible for some t and v if $E(t, v)$ denotes the empty set, and the second form (6) is not feasible if $Q(t, v, x')$ is false. This means that the guard of an event could effectively be stronger than specified if the guard was true in some state but the assignment not feasible. Such implicit specification quickly leads to models that are difficult to comprehend.

Solution For each event its feasibility (11) must be proved. Note, that for deterministic assignments the proof of feasibility is trivial (one-point rule). Also

note, that feasibility of the initialisation of a machine yields the existence of an initial state of the machine. It is not necessary to require an extra initialisation theorem as used, e.g., in Z [20].

3.4 Nondeterministic Assignments

Problem When generating proof obligations such as (12) or (15) we use for each variable two names, an unprimed name to refer to the before-state and a primed name for the after-state of the action. In classical B nondeterministic assignments were denoted by

$$x :| \, Q(t, y, x_0, x) \quad , \tag{19}$$

where x_0 denotes the value of x in the before-state and y refers to the before-state for all other variables. This notation requires renaming x_0 into x and x into x' in the proof obligations. We want to avoid renaming of variables as much as possible in order to improve readability of the proof obligations. Furthermore, note the notational inconsistency of subscripting some before-state names (x_0, for variables that may be changed by the assignment) but not others (y, for variables that are not changed by the assignment). This notation is traditionally used with predicate transformers, e.g., [16].

Solution The problem is solved easily by writing on the right-hand side of (19) a before-after predicate. Then the problem of renaming disappears, as well as the notational inconsistency. This explains the notation (6) used in Event-B. With this notation the predicate $Q(t, v, x')$ is copied without change into proof obligations, see (9). This notation follows the style of operation specifications in Z [20].

3.5 Witnesses

Problem In Section 3.2 we say that separate proof obligations are generated corresponding to the labelled elements (provided by the user), e.g., events or invariants. However, this is not directly possible for proof obligation (15):

$$\vdash \begin{array}{l} I(v) \ \wedge \ J(v, w) \ \wedge \ H(u, w) \ \wedge \ \boldsymbol{T}(u, w, w') \\ (\exists t \cdot G(t, v) \ \wedge \ (\exists v' \cdot \ \boldsymbol{S}(t, v, v') \ \wedge \ J(v', w'))) \quad . \end{array} \tag{15}$$

The two existential quantifiers in the succedent stop us from decomposing it into more manageable pieces.

Solution As mentioned in Section 2.2 proof obligation (15) describes a *simulation* of the concrete event by the abstract event. This is an intuitive concept, i.e. we have an idea of how the simulation "works". In other words, the required instantiations of the existentially quantified variables are well-understood. These can be specified as *witnesses* in an Event-B model rather than being elaborated during proof. Let $t = E(u, v, w, w')$ be the witnesses for the local variables t, and $v' = F(u, v, w, w')$ be the witnesses for the global variables v' corresponding to the after-state.

By a witness we usually understand an expression to replace one of the existentially quantified variables. But the technique can easily be generalised to predicative witnesses, i.e., by providing a predicate $P(x, u, v, w, w')$ for a variable x, where x stands for t or v'. In this generalisation the witness is not defined by an equation as previously. It just has to satisfy the less restrictive predicate $P(x, u, v, w, w')$. Of course, a predicative witness must not be void and, as a consequence, gives rise to a new proof obligation

$$\vdash \begin{array}{l} I(v) \ \wedge \ J(v,w) \ \wedge \ H(u,w) \ \wedge \ \boldsymbol{T}(u,w,w') \\ (\exists x \cdot P(x,u,v,w,w')) \quad , \end{array}$$

resembling feasibility described in Section 3.3 above.

Whatever the means by which witnesses have been specified, simple or predicative: once the two existential quantifiers have been eliminated by instantiation, we can split the proof obligation into three larger blocks for the guard $G(\ldots)$, the abstract before-after predicate $\boldsymbol{S}(\ldots)$, and the glueing invariant $J(\ldots)$. From there we can continue as described in Section 3.2, further decomposing the proof obligation. For instance, we obtain several (named) proof obligations for invariant preservation by splitting $J(\ldots)$:

$$\vdash \begin{array}{l} I(v) \ \wedge \ J(v,w) \ \wedge \ H(u,w) \ \wedge \ \boldsymbol{T}(u,w,w') \\ J_k(F(u,v,w,w'),w') \quad . \end{array} \tag{20}$$

By using witnesses in models, a part of the proof has been moved into modelling itself. The price to pay is that one has to think about proving while modelling. We do not see this as a problem because we think that modelling and proving should not be considered different activities. Note, that providing witnesses is a constructive proof technique. In modelling we prefer this over non-constructive techniques where the exact nature of the refinement relationship of the two events is left undetermined. The aim of modelling is always to increase our understanding of a model.

Using simple techniques and conventions, most of the witnesses used in (20) can be determined automatically. This frees us in practice from having to search for many witnesses.

For the *global variables* x, of an abstract machine, that are linked to the global variables y of a concrete machine by an equality invariant $x = y$, the instantiation is trivial using the one-point rule. In practice, equality invariants are assumed to hold whenever variable names are reused during refinement. Then such a variable z of the abstract machine is renamed into, say, $z1$ and variable z of the concrete machine is linked to $z1$ by the invariant $z = z1$.

For *local variables* of events we do not have invariants that we could use to find witnesses. However, most instantiations appear in practice for local variables. So we need an efficient and simple way to find witnesses for local variables. To this end, we introduce the following *convention*: when a local variable name ℓ is used in a concrete and a corresponding abstract event, then the abstract ℓ is instantiated with the concrete ℓ. Conceptually, we treat instantiation of local variables similarly to that of global variables.

3.6 Programming versus Modelling

In this short section we discuss some general points about modelling that predetermine some of the choices we have made in Event-B (those presented in Sections 3.7 to 3.10). Our discussion contrasts *modelling* and *programming* because in computing science modelling is often understood to be a form of programming. We propose to see them as activities of different nature with different aims.

The most important characteristic of a program is that it can be executed to perform some computation. When we conceive a model, we do not think about execution. We do not even require that it could be executed. How would we execute "pressing a button" when this is supposed to be done by a person in a model we have developed. It is impossible to do this and it was never our intention. In Event-B a model usually contains elements with such characteristics because we usually *include the environment* of a computing artefact, if we are developing one. In fact, nothing in Event-B requires that a model has anything to do with computation at all.

Modelling is much more concerned with observation of a model as transitions between its states occur and with reasoning about properties of the model. Models often are already useful when they are still very abstract by helping us to understand the system being modelled. The major concern in programming is execution. Properties of programs are usually directly linked to the implementation. They do not capture the system as a whole. Accordingly, programming offers a lot of support for expressing how something is computed.

Modelling is difficult without refinement. The amount of detail in a complete model of a complex system is too high to be written in a single model. In fact, we can write such models but, in practice, we cannot reason about them anymore. Refinement solves this problem by allowing us to introduce gradually more and more detail, reasoning at each refinement step about the so-enriched model. Programming usually begins with a large amount of detail necessary for the implementation of a program. Hence, it is too late to reason about it if the program is of high complexity.

Programming is associated with programming notations that have many construct convenient for programming. They are based on the assumption that the program is to be executed and that the development begins with the implementation of the program. None of this holds for modelling. We certainly do not want to begin by implementing something and we do not want our models to be restricted to those that can be executed. Concluding, we are not interested in supporting programming notation. In addition to this general discussion Sections 3.7 to 3.10 present some more technical points about constructs otherwise customary in programming.

3.7 Sequential Composition

In addition to the general problem discussed in Section 3.6 there are some technical problems we encounter with sequential composition. While modelling we usually learn about the system we are modelling. For this reason we frequently

have to switch back and forth between a model and its associated proof obligations. This switching should be as effortless as possible in order to focus on learning and on improving the model instead of analysing proof obligations with respect to their significance for the model. We cannot achieve this when we use sequential composition.

Problem Sequential composition can make proof obligations difficult to understand. We give two little examples to motivate the problem.

Assume we have an invariant $I(x, y)$ and we want to verify that the program

$$x := E(x, y) \; ; \; y := F(x, y) \tag{21}$$

preserves the invariant. Using a standard definition of sequential composition (e.g., [1,16]), we derive the following proof obligation

$$\vdash \begin{array}{l} I(x, y) \\ I(E(x, y), F(E(x, y), y)) \end{array} \quad .$$

The succedent is the invariant $I(x, y)$ rewritten according to (21). To understand the proof obligation we have to trace backwards through (21). This quickly increases the difficulty of proof obligations. In the case of (21) this is simple. Using many assignments in sequence, the problem gets more and more difficult. The following example demonstrates this on a more concrete example. In addition, it makes use of a non-deterministic assignment which aggravates the problem.

Let $x \in \mathbb{Z}$ and $y \in \mathbb{Z}$ be two integer variables; let program P be defined by:

$$\begin{array}{lll} P & \widehat{=} & \mathsf{begin} \\ & & \quad x := y - 1 \; ; \\ & & \quad y :\in \{x + 1, x - 1\} \; ; \\ & & \quad x := y * x \; ; \\ & & \quad y := y * y - x \\ & & \mathsf{end} \end{array}$$

Suppose we have specified invariant (22) that relates x and y:

$$x + y = x * y \tag{22}$$

The proof obligation to verify that (22) is an invariant of P would be a sequent as shown below[2]:

$$\vdash \begin{array}{l} x \in \mathbb{Z} \quad \wedge \quad y \in \mathbb{Z} \quad \wedge \quad y_1 \in \{y, y - 2\} \quad \wedge \quad x + y = x * y \\ y_1 * (y{-}1) + (y_1 * y_1 - y_1 * (y{-}1)) = y_1 * (y{-}1) * (y_1 * y_1 - y_1 * (y{-}1)) \end{array}$$

Note, that we had to rename y as a consequence of the appearance of the non-deterministic assignment. Now we have to judge whether this sequent is true or false. We can use a theorem prover to help us. If we think it does not hold we have to change the program. But even in this simple case it is not obvious what change in the program would cause the desired change of the proof obligation.

[2] The proof of the claim is much easier once the following three consequences of (22) are used: $x \in \{0, 2\}$, $y \in \{0, 2\}$, and $x = y$.

One could suggest that an automated theorem prover should discharge this proof obligation, should it be true. Unfortunately, there is no decision procedure for general arithmetic expressions. So this will not work.

When modelling we usually encounter sequents that are not provable because we rarely get a model correct the first time. As a consequence, we expect that we get sufficient support for improving the model. The best way to achieve this is an immediate correspondence between the model we write and the proof obligations that result from it.

Solution Event-B does not contain sequential composition. This does not mean, however, that we cannot model sequential programs in Event-B [2].

3.8 Conditional Statements

We present a problem of technical nature caused by conditional statements besides the problems discussed in Section 3.6.

Problem The greatest problem with conditional statements in refinement is that we cannot avoid generating superfluous proof obligations. Worse, these proof obligations are often difficult to understand. Let $x \in \mathbb{Z}$ be an integer variable, $a \in \text{BOOL}$ and $b \in \text{BOOL}$ boolean variables, and $m \in \mathbb{Z} \rightarrow \mathbb{Z}$ a total function. Furthermore, let program P be defined by:

$$
\begin{array}{lcl}
P & \widehat{=} & \mathsf{if}\ a = \text{FALSE} \ \wedge \ b = \text{FALSE}\ \mathsf{then} \\
 & & \quad x \ := \ m(x+1) \\
 & & \mathsf{else} \\
 & & \quad x \ := \ m(x-1) \\
 & & \mathsf{end}
\end{array}
$$

We carry out a simple data-refinement of P by program Q defined below:

$$
\begin{array}{lcl}
Q & \widehat{=} & \mathsf{if}\ (1-A)*(1-B) = 0\ \mathsf{then} \\
 & & \quad x \ := \ m(x-1) \\
 & & \mathsf{else} \\
 & & \quad x \ := \ m(x+1) \\
 & & \mathsf{end}
\end{array}
$$

where $A \in \{0,1\}$ and $B \in \{0,1\}$ are two integer variables refining a and b, respectively, using the glueing invariant[3]

$$a \ = \ \text{bool}(A = 1) \ \wedge \ b \ = \ \text{bool}(B = 1) \quad .$$

Even in this simple example it can not immediately be seen that the refinement proof obligation below has a contradictory hypothesis. The situation is much

[3] The notation bool(P) is used to denote the boolean value corresponding to truth or falsehood of predicate P.

worse when more realistic programs are considered.

$$\begin{array}{l} a \;=\; \mathrm{bool}(A = 1) \\ b \;=\; \mathrm{bool}(B = 1) \\ \neg((1 - A) * (1 - B) = 0) \\ \neg(a = \mathrm{FALSE} \wedge b = \mathrm{FALSE}) \\ \vdash \\ m(x + 1) = m(x - 1) \end{array}$$

In particular, note that we have to discharge a proof obligation that is completely insignificant with respect to the refinement relationship of P and Q. Had we used a case-statement with 10 branches, 90 out of 100 proof obligations would have been of this kind. We cannot solve this problem because it is in general not decidable which branches are supposed to refine which.

Solution Event-B does not contain conditional statements. One could suggest to name the different branches of the conditional statement and specify the refinement relationship. But this would effectively remove the conditional statement. In fact, it corresponds to the approach chosen in Event-B where each branch would correspond to a separate event. The conditional statement is not essential for the development of sequential programs in Event-B [2].

3.9 Undefinedness

Problem Any model may contain expressions that are *conditionally defined*, e.g., $1 \div x$ which is not defined for $x = 0$. A detailed discussion of this problem can be found in [7,12].

Solution In our quest for simplicity we prefer not to deviate from classical logic which has the additional advantage of being in wide-spread use. In Event-B well-definedness of expressions is treated on the level of type-checking. Type-checking works in two passes. The first pass checks whether expressions are correctly typed independently of whether they are defined everywhere. The second pass creates well-definedness proof obligations that must be discharged by proof. This technique is similar to predicate sub-typing described in [18].

3.10 Parameterisation

Problem Often a model depends on a number of parameters, e.g., the number of components in a distributed system or the size of some buffer. We do not want to write a new model each time we need different parameter values. In programming notations, e.g. Ada [21], parameterisation (also called "genericity") is used to choose specific implementation types and constants left open for customisation. This permits the development of libraries that can be reused by instantiating the parameters appropriately. For a modelling method this approach is not appropriate because the reuse is catered for execution whereas we need reuse catered for reasoning. In algebra a different form of instantiation is used. For instance, we first develop group theory and then instantiate groups with geometric transformations. Once we have proved that the transformations

form a group, we can reuse everything we have proved about groups for transformations. This technique has been adopted in algebraic specification notations, e.g. CASL [9].

Solution In Event-B parameterisation is algebraic. Event-B provides carrier sets and constants that are contained in contexts. Carrier sets and constants can be instantiated. After the axioms of the context have been proven to hold for the instantiated carrier sets and constants all theorems that have been derived from them can be reused. Machines that reference a context are parameterised by that context [6].

3.11 Openness

Problem Devising a formal method requires a lot of foresight. We would like the method to be used for years to come, estimating where it could be useful and making reasonable restrictions on the development processes in which it would be used.

Solution Being pessimistic about our capacity to predict the future and the ability to dictate changes, radical or not, to industries that could profit from the method, we choose not to finalise Event-B. We expect it to evolve according to the different needs and application domains. We propose an approach where the method from which we depart is open with respect to extensions and even changes. Still, when extending the method great care should be taken not to complicate the existing theory. In order to be able to serve a larger community duplication of concepts should be avoided and each single concept should have a simple and unambiguous interpretation.

4 Conclusion

It took a considerable amount of time to make many of the decisions presented in Section 3. We believe this effort will pay off in terms of the ease with which Event-B can be used and taught. We have not presented all decisions we have made, in particular, with respect to the notation used for predicates and expressions. In comparison to classical B their syntax has been simplified considerably. The improvements of the notation used for predicates and expressions has much less to do with constraints imposed by the need of tool support than with legibility. For a discussion of notational conventions for predicates and expressions see also [14].

We believe it is important to make the reasoning underlying the notation publicly available. This is particularly true in the light of Section 3.11. We hope that all extensions to Event-B will be made cautiously so that the notation keeps its simplicity and a lot of notation and associated methodology can be shared between different communities.

At ETH Zurich the RODIN modelling platform for Event-B is being developed that implements the techniques presented in this article. A description of the RODIN platform is published separately [3].

Acknowledgement

This article reports on results from discussions about Event-B that took place over several months with at least one meeting per week. The other two participants were Jean-Raymond Abrial and Laurent Voisin, both also at ETH Zurich.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. CUP, 1996.
2. J.-R. Abrial. Event based sequential program development: Application to constructing a pointer program. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *LNCS*, pages 51–74. Springer, 2003.
3. J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM*, volume 4260 of *LNCS*, pages 588–605. Springer, 2006.
4. J.-R. Abrial and D. Cansell. Click'n'Prove: Interactive Proofs within Set Theory. In *TPHOL*, volume 2758 of *LNCS*, pages 1–24, 2003.
5. J.-R. Abrial, D. Cansell, and D. Méry. A mechanically proved and incremental development of IEEE 1394 tree identify protocol. *FAC*, 14(3):215–227, 2003.
6. J.-R. Abrial and S. Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. *Fundamentae Informatica*, 2006.
7. J.-R. Abrial and L. Mussat. On using conditional definitions in formal theories. In D. Bert et al., editor, *ZB*, volume 2272 of *LNCS*, pages 242–269, 2002.
8. Jean-Raymond Abrial. Event driven system construction, 1999.
9. E. Astesiano, M. Bidoit, B. Krieg-Brückner, H. Kirchner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL - the common algebraic specification language. *TCS*, 286:153–196, 2002. Special issue on Abstract Data Types.
10. R.-J. 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 *LNCS*, pages 67–93. Springer, 1989.
11. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer et al., editor, *FMCO 2005*, volume 4111 of *LNCS*, 2006.
12. P. Behm, L. Burdy, and J.-M. Meynadier. Well defined B. In D. Bert, editor, *B'98: 2nd Int. B Conference*, volume 1393 of *LNCS*, pages 29–45. Springer, 1998.
13. M. Butler. csp2B: A practical approach to combining CSP and B. *FAC*, 12(3):182–198, 2000.
14. E. W. Dijkstra. The notational conventions I adopted, and why. Technical Report EWD1300, University of Texas, 2000.
15. Stefan Hallerstede. Parallel hardware design in B. In D. Bert et al., editor, *ZB*, volume 2651 of *LNCS*, pages 101–102. Springer, 2003.
16. Carroll Morgan. *Programming from Specifications: Second Edition*. PHI, 1994.
17. C. Morgan, T. Hoang, and J. Abrial. The Challenge of Probabilistic Event B. In H. Treharne at al., editor, *ZB*, volume 3455 of *LNCS*, pages 162–171. Springer, 2005.
18. J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Trans. Soft. Eng.*, 24(9):709–720, 1998.
19. S. Schneider and H. Treharne. CSP theorems for communicating B machines. *FAC*, 17(4):390–422, 2005.
20. J. M. Spivey. *The Z Notation: A Reference Manual*. PHI, 2nd edition, 1992.
21. S. T. Taft and R. A. Duff, editors. *Ada 95 Reference Manual*. Springer, 1997.
22. J.Woodcock and J.Davies. *Using Z. Specification, Refinement, and Proof*. PH, 1996.